

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。



OpenStack Cluster HA
Deployment and Operation

OpenStack 高可用集群 (下册) 部署与运维

山金孝 著

国内OpenStack领域一线技术专家在IBM和招商银行等大型企业多年项目经验总结，多位云计算技术专家联袂推荐

立足于生产环境，从原理、架构、部署、运维4个维度为构建高可用OpenStack集群提供了完整解决方案



机械工业出版社
China Machine Press

这是一部从原理、架构、部署、运维4个方面系统、深入讲解如何构建高可用OpenStack集群的著作，在理论和实践两个维度为构建高可用OpenStack集群提供了完整的解决方案。

本书从OpenStack终端用户的角色出发，以面向生产系统的OpenStack高可用集群建设为主线，对OpenStack高可用集群的原理和架构进行了深入的剖析，对部署和运维OpenStack高可用集群所依赖的各个技术栈和核心组件进行了详细的讲解。此外，书中还对Ceph和Docker等技术与OpenStack的结合应用进行了详细讲解，尤其是对Kolla项目的介绍，是本书的一大技术特色。

本书分为上下两册：

上册（第1~10章）从理论的角度剖析了OpenStack高可用集群的原理与架构。

架构篇（第1~2章）：介绍了通用云计算参考架构的设计、传统IT架构的高可用设计、云环境下的高可用设计，以及OpenStack高可用集群的架构设计。

原理篇（第3~10章）：首先详细讲解了实现OpenStack高可用集群所必须的集群资源管理器、负载均衡器、消息队列、缓存系统和数据库等OpenStack生态圈的基础技术和高可用软件；其次讲解了OpenStack的计算、网络和存储三大核心组件，以及Ceph的架构设计和使用配置。

下册（第11~15章）从实战的角度讲解了OpenStack高可用集群的部署与运维。

部署篇（第11~12章）：讲解了OpenStack基础架构软件 and 核心组件的高可用部署与实现。全面讲解了OpenStack高可用集群的落地实施过程，并将OpenStack高可用集群的部署进行了代码自动化实现，代码具有稳定的可重现性。

运维篇（第13~14章）：总结了OpenStack高可用集群运维的实践。详细讲解了基于Pacemaker高可用集群的运维，深入分析了Nova实例的高可用和Neutron网络，以及Ceph集群的运维。

拓展篇（第15章）：介绍了基于Docker的OpenStack容器化部署项目Kolla，通过Kolla实现OpenStack容器化部署。

云计算与虚拟化技术丛书



OpenStack Cluster HA
Deployment and Operation

OpenStack 高可用集群

(下册)

部署与运维

山金孝 著



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

OpenStack 高可用集群 (下册): 部署与运维 / 山金孝著. —北京: 机械工业出版社, 2017.9
(云计算与虚拟化技术丛书)

ISBN 978-7-111-58095-9

I. O… II. 山… III. 计算机网络 IV. TP393

中国版本图书馆 CIP 数据核字 (2017) 第 233151 号

OpenStack 高可用集群 (下册): 部署与运维

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 李 艺

责任校对: 李秋荣

印 刷: 北京诚信伟业印刷有限公司

版 次: 2017 年 9 月第 1 版第 1 次印刷

开 本: 186mm×240mm 1/16

印 张: 22.75

书 号: ISBN 978-7-111-58095-9

定 价: 79.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

Foreword 序1

开源项目 OpenStack 经过 7 年的发展，其社区规模日益扩大，使用用户日益增多，在云计算领域和市场的影响力也是不断增长。OpenStack 进入我国不算太晚，但由于语言、时差与开源思维等因素的影响，初期在华人用户和社区的发展稍逊于西方国家，特别是落后于美国。随着 OpenStack 日益成熟，以及我国积极倡导开源精神和自主创新精神，OpenStack 在这几年慢慢出现“西退东进”的趋势。即在西方激烈而残酷的行业竞争后，对 OpenStack 的投资有着慢慢趋缓甚至减退的迹象，而在我国，却有越来越多企业加入到 OpenStack 这个大家庭里，有越来越多的行业和大型企业选择并应用 OpenStack，也有越来越多的中国开发人员为 OpenStack 社区贡献代码，为它的成熟与发展发挥了突出的作用。

随着国内 OpenStack 市场的蓬勃发展，关于介绍 OpenStack 的书籍层出不穷。有些主要是国外经典著作的翻译版，以消除国内 OpenStack 爱好者在语言方面的障碍，但更多是来自那些经过多年探索 and 研究的国内 OpenStack 先行者们的原创作品。他们总结自己的学习成果，分享实践领域的经验教训。本书作者山金孝正是众多先行者中的一员，他参考了诸多社区公开的中英文资料，学习 OpenStack 峰会视频和讨论文档，借鉴专家们的技术分享，并结合自己工作的实践，终于著成本书。

我很荣幸在本书即将出版之际，拿到了一份电子简稿，并通读了一遍。它与其他书籍的不同之处在于，所有技术的阐述都是围绕高可用性展开的。OpenStack 的部署实际上包含了很多软件的使用，涉及面广，除了自身几个关键服务之外，它还涉及很多其他不在 OpenStack 项目里的软件和组件。如何把这些软件使用起来，配置好各种参数，使它们配合工作，从各方各面形成完整的高可用 OpenStack 部署方案，是本书的核心，也是对市场上现有各类 OpenStack 书籍的重要补充。相信读者和 OpenStack 爱好者通过本书的学习，能了解 OpenStack 原生的高可用性基础架构，熟悉 OpenStack 各相关领域开源技术和软件的应用，熟练掌握高可用 OpenStack 生产项目实施和运维。

——王庆 OpenStack 基金会个人独立董事 /
英特尔开源技术中心云计算和网络部研发经理

序2 Foreword

作为开源云计算的事实标准，OpenStack 得到了诸多厂商和企业用户的支持，越来越多的传统 IT 企业和云创公司正在将 OpenStack 作为企业发展的战略方向。全世界范围内的通信、科研、金融和电力等大中型企业用户都在成为 OpenStack 的超级用户，与此同时，对于计划进行云计算建设的企业而言，OpenStack 也是不可或缺的首要参考选项。纵观云计算这些年的发展史，在开源云计算这场 IT 技术革新的硝烟中，OpenStack 已成为最后的赢家，并且正在主导私有云领域的发展方向。

对于大多数 OpenStack 用户，尤其是中小企业用户，OpenStack 组件复杂、理论知识广泛、部署运维门槛较高和不完善的原生高可用建设支持都是阻碍 OpenStack 在企业用户，特别是传统企业用户中落地部署的主要原因。而对于像金融、证券等企业用户而言，面向生产系统的 OpenStack 集群必须具备服务的高可用性，同时要能够满足企业 7×24 的业务连续性需求，因此建设具备高可用性的 OpenStack 集群，是企业选用 OpenStack 部署云计算必须跨过的技术门槛。

本书从 OpenStack 终端用户的角色出发，以面向生产系统的 OpenStack 高可用集群建设为主线，对部署 OpenStack 高可用集群所依赖的各个基础技术栈和 OpenStack 核心组件进行了详细的原理讲解，并以实战部署的形式演示了 OpenStack 高可用集群部署的过程。此外，书中还对 Ceph 和 Docker 等技术与 OpenStack 的结合应用进行了详细讲解，尤其是 Kolla 项目的介绍，都是本书的一大技术特色。目前国内仍然缺乏一本讲解 OpenStack 高可用部署的书籍，本书填补了这一空白，相信也是很多苦于寻求 OpenStack 高可用建设方案用户的福音。

——陈沙克 九州云信息科技有限公司副总裁

Foreword 序3

在受邀为金孝兄的这本书作序时，我是怀着无比激动和忐忑的心情的：一方面为曾经共同在 IBM 战斗过的兄弟能够在开源与云计算环境中有如此专业的经验分享而激动，另一方面其实担心同样作为工程师、架构师的自己不善言语，无从下笔。但当读罢此书时，却也真心觉得想要写些什么，来感慨此书的一些出众之处。

“一切皆服务”是云计算的最终奥义，我一直这样认为。云计算从来都不是一个简单的技术命题，而是一种“让非专业人员也能够自如消费 IT 资源”的手段，是一种资源分配模式的变革。就如同社会从自然经济形态进化到商品经济形态一样，通过云计算，消费者不用再关注于从底层 IT 基础环境到顶层业务实现的全 IT 栈，而是仅需要在云生态环境中选取所需的各类 Infrastructure、Platform、Software 服务来实现自身的业务目标即可。云计算的出现将原本自产自销、纷繁多样的 IT 资源经过标准化、服务化成一种供用户购买的“商品”，从而极大地带动了依托于 IT 资源的各类新兴产业爆发式的发展，诸如互联网+、O2O、物联网等。

这是一本有明确目标的书。我阅读过很多关于云计算、云平台及 OpenStack 的相关书籍，有讲战略的，有讲业务的，有讲技术的，有讲开发的，但大多数都是为了云而云，为了技术而技术。本书却并没有开篇就与读者讨论高可用的模式、集群的实现手段等，而是从“企业为何要进行云计算建设”入手，带着明确的需求与目标来讨论“高可用”在云计算生产环境中的重要地位，从而引出需要考虑的 HADR 关键点，逐步带入实现高可用的技术手段及其原理和实战，使得读者在通读本书的过程中能够带着明确的目标进行研读，提高了读者阅读的兴趣。

这是一本站在巨人肩膀上的书。本书具备严密的逻辑性，这与作者多年来在 IBM 作为一线技术工作者的经历和经验是分不开的。从书中的许多地方都能够透出 IT 界黄埔军校——IBM 的影子，从开篇明义的 CCRA 方法论，到对 HADR 的理解与实现，均站在 IBM 这个蓝色巨人的肩膀上依托于 IBM 的架构思维来审视，加之融合了作者许多成熟经验的实战讲解，使得这本书在整个阅读的过程中能够给读者带来非常连贯、完整和丝丝入扣

的阅读体验。

这是一本实用性卓越的工具书。从事 OpenStack 相关工作的人都很清楚，OpenStack 是一个以功能为目标的项目群，旨在实现越来越全面的云平台功能，而把其本身作为企业生产环境的各非功能目标部分交由用户自行实现，其中最重要的就是高可用部分。而现实情况是，除了 Google、Amazon 这些公有云巨头外，大多数想要使用 OpenStack 作为私有云或混合云的中小企业并不具备全面的规划设计能力，从而无法顺利的将 OpenStack 应用于企业环境。本书恰恰填补了这一空白，从入门的高可用知识，到各类高可用工具的介绍，随后通过高可用集群的部署实践，指导读者一步步完成 OpenStack 高可用的构建，对中小企业将 OpenStack 生产化、商用化起到非常巨大的帮助。

如果你只是想学习 OpenStack 到底为何物，能干什么，我不推荐这本书，你可以上社区进行充分学习，比任何人著作的任何书籍都有帮助；但如果你想将 OpenStack 切实地应用起来，而不是纸上谈兵、简单玩玩而已，那我非常推荐这本书，它能够极大地减少你在浩瀚互联网海洋中摸爬滚打的时间，有的放矢、逻辑严密地指导你一步一步将一个社区化、功能化的 OpenStack 平台，构建成稳定的、高可用的企业级生产环境 OpenStack 云平台。

——行俊楠 云计算资深架构师

Foreword 序4

随着 IT 技术的不断演进以及商业模式的变革式发展，我们今天面对的技术和商业行为越来越体现出开放、敏捷与共享的特征，而云计算恰恰是这其中的基石。依托于云计算给企业带来的不同于传统 IT 架构的更加开放、灵活与共享的能力，企业可以更加敏捷地组织、实现和运营业务，从而更快速、更有效地面对现今的商业竞争。徜徉云计算的海洋中，伴随着诸多 IT 巨头以及不断涌现的创业公司的加入和推广，围绕 OpenStack 的生态圈越发蓬勃，同时也不断扩展到不同的行业、企业和业务场景中。OpenStack 已经成为开源领域云计算的事实标准，并不断推动开源云计算的发展。

与此同时，随着 OpenStack 开源分享精神与企业行业应用的深入结合，如何更加合理地将开源技术与企业的个性化场景结合？如何使得社区共建的开源能力能够满足企业级服务的效率要求？尤其是如何向企业级应用场景提供高可用的解决方案？这些都是开源精神与技术真正能够走入企业级市场必须面临的挑战。在过去面对诸多的企业级需求场景中，我曾经不断面临这些挑战，也深深地感受到在浩瀚的开源技术与信息中，要能够发掘出满足不同业务场景、体现不同个性化需求、解决独特架构体系下的技术难点，是多么的复杂与艰辛！面对这爆发式的信息浪潮，人们对技术的渴求已转为对所获取技术的质量和效率诉求，大量未经验证和无法确认的交流、分享在互联网上如瘟疫一般大肆传播于各大论坛、引擎，读者对其验真、确认的过程犹如沙海淘金，收效甚微。

山金孝先生历时经年，在大量的研究中去其糟粕，取其精华，凝其专业经验与开放信息而著此书，让我不禁为之大为赞叹！本书立意明确，步步以实践为核心，深入浅出地覆盖了如何构建企业高可用云计算的重要技术内容。精读之后让我对其匠人之心深感敬佩，也为我后续企业级产品建设与服务提供了非常多的启发与思考！感谢山金孝先生为 OpenStack 的推广及发展的用心与贡献！

——邹恒滨 四川汇揽熙云科技有限公司产品研发总监

前言 Preface

为什么写这本书

OpenStack 在云计算和 IT 基础架构领域的影响力已毋庸置疑，从 OpenStack 社区成立至今，短短几年时间里，其贡献者和用户已遍及全球各个地区和行业，OpenStack 已然成为开源领域云计算的事实标准。近年来，国内 OpenStack 初创企业不断涌现，华人企业和工程师在 OpenStack 社区中的影响力和贡献占比不断上升，以华为、中兴为主的传统 IT 企业在 OpenStack 社区不断提升自己的影响力，并在社区的董事决策中占有一席之地。此外，99Cloud、United Stack、Easy Stack、UMCloud、AWCloud 等 OpenStack 创业公司对 OpenStack 的贡献也处于全球领先的位置，可以说在国内云计算大潮的推动下，OpenStack 正在不断渗透并重构各个行业的 IT 架构。纵观 2016 年，可将其看成是 OpenStack 在国内企业用户中真正落地的元年，以国网、电信、移动和中海油等为主的大型国有企事业单位，以银联、邮蓄、兴业数金和众多地方商业银行为主的金融企业，以上交、西交、东南大学和人民在线、山西农业云、湖北楚天云等为主的科研政企单位，以及上汽集团、复兴医药等工业制造业都在部署或正式上线 OpenStack 私有云，当然还有更早便在使用 OpenStack 的诸多互联网企业。OpenStack 作为开放、包容的基础架构云平台，在国外更有像 AT&T、CERN、PayPal、BMW、eBay 和 Walmart 等重量级的大型用户，以及 IBM、HPE、Dell、Cisco、RedHat、Intel 和 Oracle 等 IT 巨头的参与。从全球目前的云计算环境发展趋势而言，企业借助 OpenStack 构建私有云已成为多数用户云化并重构数据中心的首要选择。

自 2010 年 OpenStack 的 Austin 版本发布以来，历经 7 年的时间，OpenStack 社区已发行了第 15 个版本 Ocata，其稳定性和功能在不断增强。同时，在 Liberty 版本中引入“大帐篷”概念后，OpenStack 以极为开放、包容的姿态不断整合已有或新生的 IT 技术，尤其是对 Ceph 存储和 Docker 容器技术的成功整合，真正显示出其在云计算大潮下中流砥柱的地位和集成引擎的强大一面。尽管在这几年的发展过程中也饱受指责、批评和诟病，同时还受到 Docker 后来居上甚至取而代之的威胁，但是 OpenStack 依然一路高歌向前，并以拥抱一切竞争对手的姿态不断完善和巩固自己的主导地位。时至今日，可以毫不夸张地说，在

云计算领域，没有 OpenStack 实现不了的功能，也没有 OpenStack 整合不了的技术，或者说在开源领域，OpenStack 已成为了云计算的代名词。

OpenStack 虽然如此火爆，但是想要实现面向生产环境的 OpenStack 高可用云计算环境却并非易事，尤其对于高度依赖传统 IT 架构的企业而言，服务高可用一直是企业部署和使用 OpenStack 难以跨越的鸿沟，而这也成为了 OpenStack 真正落地并走向普通企业用户需要解决的“最后一公里”。OpenStack 以开源共享的方式为用户走向云计算提供了便捷之道，但是其内部组件之复杂、涉及技术栈之多、版本更新之快以及部署维护之困难往往也超出很多用户的预期。围绕这些问题，OpenStack 社区孵化了很多新项目以期解决被诟病最多的入门困难、部署困难和维护困难等问题。但是为了实现 OpenStack 的自动化部署，这些项目无一不又重新引入了新的理论技术，并从其他维度增加了用户使用 OpenStack 的学习和维护成本，而且这些项目并没有很好地解决企业用户最为关心的 OpenStack 服务高可用问题。以上种种，归其原因，在于社区曾明确 OpenStack 的高可用应交由用户的基础架构软件而非上层 OpenStack 项目来实现。因此，对于 OpenStack 企业用户而言，在 OpenStack 社区上游功能交付和终端用户部署实施之间一直横亘着一个难题，即如何在使用 OpenStack 的过程中通过集成高可用基础架构软件，实现保证业务连续性的 OpenStack 高可用集群。

虽然目前各个 OpenStack 厂商都有自己的 OpenStack 高可用解决方案，但是采用厂商定制商业方案似乎有违使用 OpenStack 的初衷，而且市面上的 OpenStack 书籍多以 OpenStack 理论讲解和功能部署为主，却没有一本专门面向 OpenStack 高可用实施部署的中文书籍（虽然在 OpenStack 官方网站和其他互联网站点中也能找到 OpenStack 高可用建设的相关资料，但是这些资料或者藏头露尾，或者零散不全，且以英文资料居多）。秉承共享精神，为了给国内社区用户，尤其是新入门的 OpenStack 用户提供一套完整的 OpenStack 高可用部署参考方案，同时对 OpenStack 进行由底层基础架构软件至上层 OpenStack 核心组件的原理分析与高可用部署的一站式讲解，我们决定编写一本关于 OpenStack 高可用集群原理、部署与运维的书籍，对以 OpenStack 为核心的技术栈进行全面剖析，即从理论知识准备到高可用实战操作，再到后期的高可用集群运维进行透彻的分析和介绍，同时围绕 OpenStack 生态圈，对 Ceph 和 Docker 与 OpenStack 的集成应用进行实战讲解。希望本书能够为企业用户在 OpenStack 的应用部署中提供微薄之力。

本书的主要内容和特色

本书分为上下两册，理论与实战结合，全面讲解了 OpenStack 的技术知识点。上册讲解了 OpenStack 相关的基础架构软件，如集群管理软件 Pacemaker、负载均衡及高可用软件 HAProxy 和 Keepalived、缓存系统 Memcached 和 Redis、数据库 MariaDB 和 MongoDB 以及消息队列系统 RabbitMQ 等基础软件的理论知识，也可以了解到 OpenStack 三大核心组件——计算（Nova）、存储（Cinder/Ceph）和网络（Neutron）的架构原理及使用方式。

下册从实战角度讲解了如何对 OpenStack 的基础架构软件 and 核心组件项目进行高可用集群部署，然后介绍了如何在实际应用中，对 OpenStack 高可用集群进行运维分析与故障解决。

整体而言，本书从项目实施的角度，按照“理论基础—实战部署—后期运维”的方式进行循序渐进的讲解，围绕 OpenStack 生态圈，不仅介绍了 OpenStack 的组件项目，还对其依赖的基础架构软件进行了完整介绍，同时对当前较为热门的 Ceph 和 Docker 在 OpenStack 中的集成应用也进行了分析和实战演示。

本书面向的读者

书中以 Linux 系统运维为基础，通过 Pacemaker 集群软件为各项服务提供高可用性，涉及负载均衡、数据库、缓存系统、消息队列和云计算领域的存储、网络以及计算资源虚拟化等诸多知识点，是一本整合了 OpenStack 生态圈的技术书籍，非常适合 OpenStack 入门初学者、运维工程师和 Open-Stack 高可用架构工程师及 Ceph 存储管理员等从业人员阅读。

此外，本书也适合高校在读本科生或研究生用作 OpenStack 研究、部署和实践的参考书。通过对本书的学习，读者将可以从 OpenStack 入门级别走上高可用 OpenStack 生产项目实施和运维工程师的台阶。

如何阅读本书

阅读本书之前，读者应该具备一定的 KVM 虚拟化知识、SAN 网络或分布式存储知识以及 Linux 系统运维、集群管理和高可用相关方面的基础知识。由于篇幅较多，本书分为上、下两册，其中上册包括第 1~10 章，下册包括第 11~15 章，按照架构篇（第 1~2 章）、原理篇（第 3~10 章）、部署篇（第 11~12 章）、运维篇（第 13~14 章）以及扩展篇（第 15 章）的结构进行编排，读者如果仅关注于某个阶段的参考学习，可直接参考目录结构进入相关章节。按章节顺序，本书讲述了如下内容。

第 1 章描述了云计算建设的必要性，同时对用户在公有云与私有云之间的决策提供建设性的参考，并对企业实施云计算的进阶路线提供指引和参考架构，此外还对传统 IT 架构和云计算环境下的高可用架构设计进行了介绍。

第 2 章对 OpenStack 高可用架构的功能组件和集群核心服务项目进行了介绍，还对 Redhat 和 Marantis 两大 OpenStack 领导厂商的高可用架构进行了详细介绍，并与其他 OpenStack 厂商的高可用架构进行了对比分析。

第 3 章介绍了 OpenStack 高可用架构中的集群资源管理器 Pacemaker 的原理、架构和使用方法。

第4章介绍了 OpenStack 高可用架构中的集群负载均衡与高可用软件 HAProxy 和 Keepalived 的原理、架构和使用方法。

第5章介绍了 OpenStack 高可用集群中的消息队列系统 RabbitMQ 的原理、架构和高可用配置与使用方法。

第6章介绍了 OpenStack 高可用集群中的缓存系统 Memcached 和 Redis 的原理及使用方法。

第7章介绍了 OpenStack 高可用集群中的关系型数据库 MariaDB 和非关系型数据库 MongoDB，同时对这两种数据库的高可用配置与使用方法进行了详细介绍。

第8章介绍了 OpenStack 高可用集群中核心服务之一的计算服务 Nova 项目，对 Nova 的架构原理、使用配置以及服务高可用均进行了详细介绍。

第9章介绍了 OpenStack 高可用集群中核心服务之一的网络服务 Neutron 项目，对 Neutron 的插件式架构、高可用配置以及不同的 Neutron 网络模式及其使用方法进行了详细介绍。

第10章介绍了 OpenStack 高可用集群中核心服务之一的块存储服务 Cinder 和 Ceph 分布式存储集群，对 Cinder 块存储的架构、配置和多后端存储及其使用进行了详细介绍，同时对 Ceph 在 OpenStack 中的集成使用进行了实战演示。

第11章介绍了 OpenStack 基础架构组件的高可用部署，以实战部署的方式讲解了如何通过 Pacemaker 集群部署高可用的 OpenStack 基础服务组件。

第12章介绍了 OpenStack 核心服务组件的高可用部署，以实战部署的方式讲解了如何通过 Pacemaker 集群部署高可用的 OpenStack 核心服务组件。

第13章介绍了如何运维基于 Pacemaker 的 OpenStack 高可用集群，并对 Neutron 网络和 Nova 计算服务的高可用性进行了详细分析。

第14章介绍了 Ceph 分布式存储集群的运行维护经验，并提供了如何配置和自定义使用 Ceph 集群的参考。

第15章介绍了如何通过 Docker 容器与 OpenStack 集成项目 Kolla 进行 OpenStack 的容器化部署，同时对 OpenStack 的 Kolla 项目进行了详细介绍。

勘误和资源

在本书的写作过程中，笔者参考了很多 OpenStack 官方社区的资料和历届 OpenStack 峰会的讨论文档与视频，同时也参考了很多开源软件的官方资料和技术专家的经验分享，诚恳希望能为 OpenStack 爱好者与从业者呈现一本涵盖基础理论与高可用实战部署和运维的参考书籍。但是由于 OpenStack 社区版本变化之快，使得任何 OpenStack 相关书籍都难以跟上每年两次的 OpenStack 发行版本，加之笔者水平有限，书中难免存在技术延后和谬误观点，若书中有任何不妥之处，恳请读者批评指正。读者可将意见发送至邮箱

ynwssjx@126.com 或者通过微信公众号“OpenStackGeek”进行反馈，我们将实时跟进 OpenStack 社区的发展变化，并吸取读者的宝贵意见。另外，本书涉及的 OpenStack 高可用集群部署源代码已全部上传至 GitHub，读者可以从网站 <https://github.com/ynwssjx/OpenStack-HA-Deployment> 下载查看并参考实现。

致谢

开源共享是人类历史上最伟大的精神之一，在此向 Linux 创始人和开源精神领袖 Linus Torvalds 致敬，向 OpenStack 项目发起者 NASA 和 Rackspace 致敬，向 OpenStack 社区所有参与者和无私的代码贡献者致敬。

本书的编写历时半年有余，在工作和生活极为繁忙的阶段，笔者仍然坚持每日查阅资料 and 整理文章，期间得到了招商银行很多同事和领导的关心，同时也得到了很多 IBM 前同事和领导的支持，在此一并谢过。正是你们的关心和支持，才使得我在繁忙的工作之余仍然怀着一颗敬畏之心进行写作。

在本书的策划和写作期间，机械工业出版社华章分社的杨福川先生和李艺女士给予了极大的关心和帮助，在此感谢杨福川先生对本书的策划和李艺女士对全文的审阅校对，正是你们的辛勤付出才有了本书的问世。

另外，还要感谢我的妻子杨彩凤女士在写作期间对我生活上的照顾与理解，在多少个深夜与凌晨，正是你的理解与支持让我能全身心地投入写作中。在此也要感谢我的父母，感谢你们的默默养育和辛勤付出。在本书的写作期间，奶奶的仙逝是我最大的悲恸，谨以此书慰藉奶奶的在天之灵，您一世的慈祥我将永远铭记。

最后，感谢所有为本书的编写提供了帮助、支持与鼓励的朋友们，感谢陈沙克、刘世民、吴业亮等无私分享博文的技术爱好者，感谢所有为 OpenStack 社区无私奉献的企业和志愿者。相信在开源精神的共鸣下，OpenStack 一定会变得更好！

Contents 目 录

序 1
序 2
序 3
序 4
前言

架构篇

第 1 章 云计算架构设计及业务系统

高可用..... 2

1.1 企业为何要进行云计算建设..... 2

1.1.1 政策导向与 IT 发展的必然..... 2

1.1.2 业务导向与 IT 弹性需求..... 4

1.1.3 技术导向与 IT 自动化..... 4

1.1.4 成本导向与 TCO..... 6

1.2 企业如何决策公有云与私有云..... 8

1.2.1 云计算部署模式对比..... 8

1.2.2 如何决策私有云与公有云..... 10

1.3 云计算架构设计与进阶路线..... 13

1.3.1 云计算生态模型..... 13

1.3.2 云计算架构基本模型..... 15

1.3.3 通用云计算参考架构..... 16

1.3.4 云计算实施进阶路线..... 20

1.4 业务系统高可用性概述..... 22

1.4.1 业务系统高可用性..... 23

1.4.2 业务系统容灾恢复..... 24

1.5 传统 IT 架构高可用设计..... 26

1.5.1 传统数据中心 HADR 设计原则..... 26

1.5.2 故障划分与 HADR 高可用实现..... 27

1.6 云环境下的高可用设计..... 29

1.6.1 云计算 HADR 架构设计原则..... 30

1.6.2 云计算 HADR 架构设计实现..... 33

1.7 本章小结..... 36

第 2 章 OpenStack 高可用集群

架构概述..... 37

2.1 OpenStack 高可用集群功能组件..... 37

2.1.1 集群控制节点..... 38

2.1.2 集群计算节点..... 39

2.1.3 集群存储节点..... 40

2.1.4 集群网络节点..... 41

2.1.5 集群负载均衡器..... 43

2.1.6 集群网络拓扑..... 44

2.2 OpenStack 高可用集群服务组件..... 47

2.2.1 认证服务 Keystone..... 47

2.2.2	镜像服务 Glance	50
2.2.3	计算服务 Nova	52
2.2.4	块存储服务 Cinder	54
2.2.5	网络服务 Neutron	57
2.2.6	控制面板 Horizon	59
2.2.7	其他 OpenStack 服务	60
2.3	Redhat OpenStack 高可用部署架构	63
2.3.1	Redhat OpenStack 高可用集群部署架构	63
2.3.2	Redhat OpenStack 高可用集群服务规划	67
2.4	Mirantis OpenStack 高可用部署架构	71
2.4.1	Mirantis OpenStack 高可用集群部署架构	72
2.4.2	Mirantis OpenStack 自定义高可用集群架构	76
2.5	其他厂商 OpenStack 高可用部署架构介绍及对比分析	79
2.5.1	Juniper Networks OpenStack 高可用部署方案	80
2.5.2	HPE OpenStack 高可用部署方案	81
2.5.3	TCP Cloud OpenStack 高可用部署方案	83
2.5.4	Paypal OpenStack 高可用部署方案	84
2.5.5	Oracle OpenStack 高可用部署方案	87
2.5.6	OpenStack 高可用部署方案对比分析	87
2.6	本章小结	89

原理篇

第 3 章 集群资源管理系统 92

3.1	Pacemaker 概述	93
3.2	Pacemaker 集群分类	95
3.3	Pacemaker 集群架构	97
3.4	Pacemaker 内部组件	98
3.5	Pacemaker 集群配置信息管理	99
3.5.1	Pacemaker 集群状态信息	100
3.5.2	Pacemaker 集群配置信息	101
3.6	Pacemaker 集群管理工具 PCS	108
3.6.1	PCS 命令行工具	108
3.6.2	PCS 用户接口界面	110
3.7	Pacemaker 集群资源管理	113
3.7.1	集群资源代理	113
3.7.2	集群资源约束	118
3.7.3	集群资源类型	120
3.7.4	集群资源规则	124
3.8	本章小结	126

第 4 章 集群负载均衡系统 127

4.1	Keepalived 概述与配置	128
4.1.1	Keepalived 及 LVS 概述	128
4.1.2	Keepalived 工作原理	133
4.1.3	Keepalived 调度算法	136
4.1.4	Keepalived 路由方式	137
4.1.5	Keepalived 配置与使用	138
4.2	HAProxy 概述与配置	144
4.2.1	HAProxy 概述	144
4.2.2	HAProxy 配置	146
4.2.3	HAProxy 监控页面	151
4.2.4	HAProxy 配置参考	154

4.3 本章小结	158	6.2.1 Redis 缓存概述	204
第 5 章 集群消息队列系统	159	6.2.2 Redis 数据交换	205
5.1 AMQP 概述	160	6.2.3 Redis 数据持久化	206
5.2 RabbitMQ 概述	161	6.2.4 Redis 数据高可用	207
5.3 RabbitMQ 工作原理	167	6.2.5 Redis 高可用配置	209
5.4 RabbitMQ 基本配置	169	6.2.6 Redis 集群概述	216
5.5 RabbitMQ 集群基础	170	6.2.7 Redis 在 OpenStack 中的应用	218
5.5.1 RabbitMQ 集群概述	170	6.3 本章小结	219
5.5.2 RabbitMQ 的集群配置	171	第 7 章 集群数据库系统	221
5.6 RabbitMQ 集群管理	174	7.1 关系型数据库——MariaDB	221
5.6.1 RabbitMQ 集群节点启停	174	7.1.1 MySQL 概述	221
5.6.2 RabbitMQ 的集群节点移除	175	7.1.2 MariaDB 概述	224
5.7 RabbitMQ 的集群队列镜像	177	7.1.3 MariaDB 安装配置	225
5.8 基于 Pacemaker 的高可用 RabbitMQ 集群	181	7.1.4 MariaDB 高可用方案	233
5.8.1 Active/Passive 模式的 RabbitMQ 集群	181	7.1.5 MariaDB Galera Cluster 概述	236
5.8.2 Active/Active 模式的 RabbitMQ 集群	182	7.1.6 MariaDB Galera Cluster 配置	239
5.9 RabbitMQ 在 OpenStack 中的应用分析	187	7.2 非关系型数据库——MongoDB	249
5.10 本章小结	192	7.2.1 NoSQL 概述	249
第 6 章 集群缓存系统	193	7.2.2 MongoDB 概述	251
6.1 Memcache 缓存系统	193	7.2.3 MongoDB 安装配置	254
6.1.1 Memcache 缓存概述	193	7.2.4 MongoDB Replica Set 概述	258
6.1.2 Memcache 的工作原理	194	7.2.5 MongoDB Replica Set 部署	260
6.1.3 Memcache 的功能特点	196	7.3 本章小结	265
6.1.4 Memcache 集群概述	197	第 8 章 OpenStack 计算服务	267
6.1.5 Memcache 集群高可用	201	8.1 OpenStack 项目概述	267
6.2 Redis 缓存系统	204	8.1.1 OpenStack 项目概要	267
		8.1.2 OpenStack 版本发行	268
		8.1.3 OpenStack 组织机构	272
		8.1.4 OpenStack 使用情况	274
		8.1.5 OpenStack 服务项目	276

8.2	Nova 项目概述	277	第 9 章	OpenStack 网络服务	388
8.2.1	Nova 架构设计	277	9.1	Neutron 网络概述	388
8.2.2	Nova 功能模块	282	9.2	Neutron 网络架构	394
8.3	Nova 分区与区域	285	9.2.1	Neutron 网络架构概述	394
8.3.1	Nova 中的 Region	285	9.2.2	Neutron Plugin 与 Agent	396
8.3.2	Nova 中的 Cell	288	9.2.3	Neutron L3 Service 分析	402
8.3.3	Nova 中的 Availability Zone	292	9.3	Neutron 网络类型	408
8.3.4	Nova 中的 Host Aggregate	294	9.3.1	Provider 网络	408
8.4	Nova Hypervisor 配置概述	297	9.3.2	Self-Service 网络	411
8.4.1	虚拟化与 Hypervisor 概述	297	9.4	Provider 网络部署与分析	415
8.4.2	Nova Hypervisor 归类支持	303	9.4.1	Provider 网络基于 OpenvSwitch 实现	415
8.4.3	Nova Hypervisor 选取配置	308	9.4.2	Provider 网络基于 LinuxBridge 实现	424
8.5	Nova 主机策略	317	9.4.3	Provider 网络创建与验证	429
8.5.1	Nova scheduler 主机过滤	317	9.5	Self-Service 网络部署与高可用	433
8.5.2	Nova scheduler 主机加权	324	9.5.1	Self-Service 网络实现	433
8.5.3	Nova scheduler 配置选项	329	9.5.2	Self-Service 网络高可用	450
8.6	Nova 实例创建	333	9.6	L3 HA 高可用方案	452
8.6.1	Nova 实例创建流程	333	9.6.1	L3 HA 高可用部署实现	452
8.6.2	Nova 实例状态变更	341	9.6.2	L3 HA 高可用验证与分析	459
8.6.3	Nova 实例创建方法	347	9.7	DVR 高可用方案	470
8.7	Nova 实例迁移	354	9.7.1	DVR 高可用部署实现	470
8.7.1	Nova 实例 resize/migrate 迁移	354	9.7.2	DVR 高可用验证与分析	477
8.7.2	Nova 实例 live-migration 迁移	365	9.7.3	DVR 与 L3 HA 对比	492
8.8	Nova 实例高可用	376	9.8	DVR/L3 HA 高可用方案	493
8.8.1	Nova 实例高可用概述	376	9.8.1	DVR/L3 HA 高可用部署实现	493
8.8.2	Nova 实例高可用之 Evacuate/ Rebuild	378	9.8.2	DVR/L3HA 高可用验证与分析	499
8.8.3	Nova 实例高可用之 Pace- maker_remote	382	9.9	本章小结	511
8.9	本章小结	387	第 10 章	OpenStack 存储服务	512
			10.1	OpenStack 存储概述	513

10.1.1	OpenStack 存储分类对比	513			
10.1.2	OpenStack 存储后端选择	515			
10.2	Cinder 块存储	519			
10.2.1	Cinder 块存储架构	519			
10.2.2	Cinder 块存储使用	520			
10.2.3	Cinder 块存储插件	524			
10.2.4	Cinder LVM 插件实现	529			
10.2.5	Cinder NFS 插件实现	534			
10.2.6	Cinder Multi-Backends 实现	540			
10.3	Ceph 存储系统	545			
10.3.1	Ceph 背景概述	545			
10.3.2	Ceph 架构设计	547			
10.3.3	Ceph 工作原理	553			
10.3.4	Ceph 部署实现	559			
10.4	Ceph 集成 OpenStack	564			
10.4.1	Ceph 集成 OpenStack 概述	564			
10.4.2	Ceph 集成 OpenStack 准备	566			
10.4.3	Ceph 集成 Glance	569			
10.4.4	Ceph 集成 Cinder	571			
10.4.5	Ceph 集成 Nova	574			
10.4.6	Ceph 集成 OpenStack 验证	578			
10.5	本章小结	581			
部署篇					
第 11 章 OpenStack 高可用集群					
	基础服务部署	584			
11.1	OpenStack 集群高可用离线部署	584			
11.1.1	制作 OpenStack 离线安装 pip 源	585			
11.1.2	制作 OpenStack 离线安装 yum 源	592			
11.2	OpenStack 集群高可用部署架构 设计	599			
11.2.1	OpenStack 高可用部署实验 环境架构	599			
11.2.2	OpenStack 高可用部署生产 环境架构	603			
11.2.3	OpenStack 高可用部署软件 拓扑架构	608			
11.3	OpenStack 集群高可用部署实验 环境准备	610			
11.3.1	控制节点 VMware 宿主机 准备	611			
11.3.2	控制节点 KVM 虚拟机 准备	617			
11.3.3	计算节点 VMware 虚拟机 准备	624			
11.4	OpenStack 高可用集群基础 服务部署	625			
11.4.1	Pacemaker 集群管理软件 部署	625			
11.4.2	HAProxy 负载均衡器高可用 部署	628			
11.4.3	MariaDB 关系数据库高可用 部署	633			
11.4.4	Memcache 缓存系统高可用 部署	639			
11.4.5	RabbitMQ 消息队列高可用 部署	640			
11.4.6	MongoDB 非关系数据库 高可用部署	643			
11.5	本章小结	646			

第 12 章 OpenStack 高可用集群**核心服务部署 647****12.1 OpenStack 控制节点服务高可用****部署 647****12.1.1 Keystone 认证服务高可用****部署 648****12.1.2 Glance 镜像服务高可用****部署 655****12.1.3 Cinder 块存储服务高可用****部署 660****12.1.4 Neutron 网络服务高可用****部署 665****12.1.5 Nova API 服务高可用部署 676****12.1.6 Ceilometer 数据采集服务****高可用部署 682****12.1.7 Heat 编排服务高可用部署 687****12.1.8 Horizon 控制面板服务****高可用部署 691****12.2 OpenStack 计算节点服务高可用****部署 694****12.2.1 OpenStack 计算节点高可用****实现概述 694****12.2.2 OpenStack 计算节点高可用****方案分析 695****12.2.3 OpenStack 计算节点 Pace-****maker 高可用集群分析 696****12.2.4 OpenStack 计算节点 Pace-****maker 高可用集群实现 697****12.3 OpenStack 集群服务高可用****验证 707****12.3.1 OpenStack 高可用集群****功能性验证 707****12.3.2 OpenStack 高可用集群****高可用验证 722****12.4 本章小结 731****运维篇****第 13 章 OpenStack 高可用集群****运维最佳实践 734****13.1 Pacemaker OCF 资源代理故障****诊断分析 735****13.1.1 Pacemaker 集群 OCF 资源****代理使用介绍 735****13.1.2 Pacemaker 集群 OCF 资源****代理定义语法 737****13.1.3 Pacemaker 集群 OCF 资源****代理调试诊断 744****13.2 Pacemaker 集群调试与管理维护 749****13.2.1 Pacemaker 集群日志系统****设置 749****13.2.2 Pacemaker 集群日志构成****分析 751****13.2.3 Pacemaker 集群日志调试****分析 755****13.2.4 Pacemaker 集群 GUI 管理****界面 758****13.3 OpenStack 实例高可用原理分析****与问题诊断 765****13.3.1 OpenStack 高可用集群计算****节点资源配置 765****13.3.2 OpenStack 高可用集群****Fence_compute 分析 766****13.3.3 OpenStack 高可用集群**

NovaEvacuate 分析	771
13.3.4 计算节点高可用实现原理与 问题诊断分析	774
13.4 OpenStack Neutron 网络理解与 故障问题诊断	781
13.4.1 OpenStack Neutron 网络概念 基础	781
13.4.2 OpenStack Neutron 网络深入 理解	784
13.4.3 OpenStack Neutron 网络故障 分析	803
13.5 OpenStack 日常管理与运维	811
13.5.1 OpenStack 日志设置管理与 使用	811
13.5.2 OpenStack 故障实例数据 检查恢复	813
13.5.3 OpenStack 故障计算节点 实例恢复	816
13.5.4 OpenStack 实例间浮动 IP 地址管理	818
13.5.5 OpenStack 服务运行缓慢 解决方案	819
13.5.6 OpenStack 配置文件及数据库 备份	821
13.6 本章小结	824

第 14 章 Ceph 存储集群运维最佳 实践

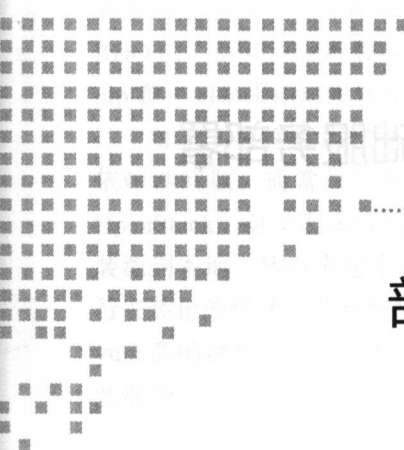
14.1 Ceph 规划配置与性能调优	825
14.1.1 Ceph 硬件配置推荐	825
14.1.2 Ceph 配置文件设置	829

14.1.3 Ceph CRUSH 自定义	843
14.1.4 Ceph SSD 应用场景	854
14.1.5 Ceph 性能调优关键	862
14.2 Ceph 运维与常见故障处理	867
14.2.1 Ceph OSD 与 PG 状态	867
14.2.2 Ceph OSD 节点增删	871
14.2.3 Ceph MON 节点增删	875
14.2.4 Ceph Journal 故障维护	877
14.2.5 Ceph OSD 故障硬盘更换	880
14.2.6 Ceph 存储节点停机维护	881
14.2.7 Ceph 容量耗尽解决方案	883
14.2.8 Ceph 常用命令使用参考	886
14.3 本章小结	891

扩展篇

第 15 章 Docker 容器部署 Open- Stack

15.1 OpenStack 与 Docker	894
15.1.1 容器与虚拟机的现状	894
15.1.2 OpenStack 融合 Docker	897
15.2 Kolla 项目介绍	900
15.2.1 Kolla 项目使命	900
15.2.2 Kolla 及其现状	905
15.2.3 Kolla 内部组件	907
15.3 Kolla 容器化部署 OpenStack	915
15.3.1 系统部署环境准备	915
15.3.2 制作 Docker 镜像	917
15.3.3 部署 Docker 容器	919
15.3.4 OpenStack 功能验证	920
15.4 本章小结	924



部 署 篇

- 第 11 章 OpenStack 高可用集群基础服务部署
- 第 12 章 OpenStack 高可用集群核心服务部署



OpenStack 高可用集群基础服务部署

部署

OpenStack 是由众多独立子项目构成的复杂开源云计算平台，这些独立子项目不仅包括 OpenStack “大帐篷”覆盖下各自发展的众多开源项目，还包括“大帐篷”外很多基础性的开源项目。纵观 OpenStack 集群的功能构成和部署过程，可以说 OpenStack 是集众多开源项目于一体的大成者，而 OpenStack 云平台的复杂性和高门槛也正源自其“开源集大成者”。本书在介绍 OpenStack 高可用集群部署时，将 OpenStack 高可用集群的部署一分为二，即基础服务的高可用部署和核心服务的高可用部署。本章将重点介绍 OpenStack 基础依赖服务的高可用部署。这些服务包括集群资源管理软件 Pacemaker、负载均衡软件 HAProxy、关系型数据库 MariaDB、缓存系统 Memcache/Redis、消息队列系统 RabbitMQ 和非关系型数据库 MongoDB，这些基础服务的工作原理和相关配置使用在前面已进行了详细介绍，本章主要针对这些基础服务在 Pacemaker 集群中的高可用部署进行介绍。

在 OpenStack 高可用集群部署到生产环境之前，建议在实验环境或测试环境中对 OpenStack 集群的高可用功能进行部署验证。本章从实验环境的角度出发，对如何部署搭建 OpenStack 高可用集群实验环境进行了介绍，并给出了 OpenStack 离线安装部署中本地离线安装源的制作方法。通过本章实验环境的准备介绍，读者可以参考并搭建出自己的实验环境并对后续 OpenStack 相关服务的高可用性部署进行验证，在高可用功能验证完成后，便可轻松将实验环境迁移至生产环境中。

11.1 OpenStack 集群高可用离线部署

OpenStack 是开源项目，用户可以通过 Internet 利用网络安装源进行实时部署，也可以下载源代码或者开源厂商发行的 RPM 安装包进行离线部署。由于网络安装源常位于国外，

且国内网络环境限制，通常致使安装过程中断或者耗时较长的情况，尤其是在大规模安装环境中，这种方式显然是不明智的。此外，对于像金融、政府等行业，由于安全限制等原因，数据中心与互联网通常是隔离的，类似的企业想要部署基于 OpenStack 的私有云，则必须通过离线安装方式来实现。离线安装方式可以是源代码下载安装方式，也可以是类似 RDO 的厂商开源发行版本离线安装方式。如果采用离线源代码安装，则通常需要配置本地 pip 源；如果采用离线软件安装包的形式安装，则通常需要配置本地 yum 源。

对于多数终端用户而言，离线源代码安装方式似乎难度太大，而且要解决很多软件依赖性问题，通常仅有对 OpenStack 具有深入理解的用户才会使用离线源代码安装部署 OpenStack。对于大多数用户，较为理想的离线安装方式便是通过互联网同步远程软件包安装源到本地，然后搭建本地共享 yum 源仓库，这样集群节点在部署安装时便可利用内网进行安装包的快速下载和安装。为了便于不同级别的用户参考，本节将介绍本地 pip 源和本地 yum 源的制作方式，用户可以根据自身情况选择和制作不同的本地源进行 OpenStack 的安装部署。

11.1.1 制作 OpenStack 离线安装 pip 源

OpenStack 目前支持主流的 Linux 操作系统，而 OpenStack 主要用 Python 语言开发，对于 Linux 操作系统而言，yum 是使用最为广泛的软件包安装工具。而在 Python 的世界里，pip 在软件包的安装和依赖解决方面有着类似 yum 的功能，因此也是使用极为广泛的软件包安装工具。在 OpenStack 源代码离线安装中，通常使用 pip 来安装 OpenStack 各个项目所需的 Python 依赖包。OpenStack 各个项目在源代码发行时，通常会将该项目所需的依赖包整理为一个文本文件，用户在安装部署该项目源代码之前，必须事先安装符合该文本文件中所需的全部依赖包，否则源代码安装将无法继续。而 pip 在安装依赖包时，默认使用的 pip 源由位于国外的 www.pypi.org 网站提供，由于国内特殊的网络环境，在批量安装依赖包时将会出现耗时较长或者安装中断等现象，因此，制作 pip 本地源变得尤为重要。本节将介绍几种制作 pip 本地源的方式，通过 pip 本地源，用户可以快速实现 OpenStack 项目依赖包的快速安装。

pip^①是 Python 中目前最为流行和强大的安装包管理工具，pip 未来将会是传统 [setuptools](http://pypi.python.org/pypi/setuptools)^② Python 软件包管理工具的替代者，虽然 [setuptools](http://pypi.python.org/pypi/setuptools) 也提供了简单易用的包安装工具，如最为常见的 `easy_install` 命令行工具，便是由 [setuptools](http://pypi.python.org/pypi/setuptools) python 包管理软件所提供的命令行，但是，[setuptools](http://pypi.python.org/pypi/setuptools) 后期将不再被维护（Python 3 将不再支持 [setuptools](http://pypi.python.org/pypi/setuptools)），因此对于 Python 包管理而言，pip 将是最佳选择。通常在安装 pip 之前，需要安装 [setuptools](http://pypi.python.org/pypi/setuptools)，如果是 Python 3 环境，则需要先安装 [Distribute](http://pypi.python.org/pypi/Distribute)。对于 CentOS 6 或 RHEL 6 以上的 Linux 操

① pip 最新安装包下载地址：<http://pypi.python.org/pypi/pip#downloads>。

② [setuptools](http://pypi.python.org/pypi/setuptools) 安装包下载地址：<http://pypi.python.org/pypi/setuptools>。

作系统，可以通过如下命令来安装适合用户当前环境的 `setuptools`：

```
wget https://bootstrap.pypa.io/ez_setup.py -O - | python
```

或者通过 `yum` 来安装 `setuptools`：

```
yum install setuptools
```

`setuptools` 安装完成后，即可使用 `easy_install` 包安装管理工具。使用 `easy_install` 安装 `pip` 的命令如下：

```
easy_install pip
```

除了提供 Python 软件包安装功能之外，`pip` 还提供了类似 `yum` 的安装包查询、卸载和搜索等功能，默认情况下 `pip` 使用 `www.pypi.org` 提供的资源。`pip` 的使用方法可以通过 `help` 命令来查看：

```
root@mitaka ~]# pip help
Usage:
  pip <command> [options]
```

Commands:

<code>install</code>	Install packages.
<code>uninstall</code>	Uninstall packages.
<code>freeze</code>	Output installed packages in requirements format.
<code>list</code>	List installed packages.
<code>show</code>	Show information about installed packages.
<code>search</code>	Search PyPI for packages.
<code>wheel</code>	Build wheels from your requirements.
<code>help</code>	Show help for commands.

...

如使用 `pip` 安装 `flask`：

```
[root@mitaka ~]# pip install flask
```

查看已经安装的包：

```
[root@mitaka ~]# pip show flask
```

Name: Flask

Version: 0.11.1

Summary: A microframework based on Werkzeug, Jinja2 and good intentions

Home-page: <http://github.com/pallets/flask/>

Author: Armin Ronacher

Author-email: armin.ronacher@active-4.com

License: BSD

Location: `/usr/lib/python2.7/site-packages`

Requires: `click, Jinja2, Werkzeug, itsdangerous`

`pip` 在配置文件 `~.pip/pip.conf` 中通过 `index-pypi` 参数配置了默认安装源，默认安装源为 `http://pypi.python.org/simple`，因此 `pip install` 和 `pip search` 命令默认都会到 `pypi.python`。

org 上下载或搜索 pypi (python package index) 软件包。相对国外 pip 源, 国内网络环境要快很多, 因此如果采用网络安装源进行 pip 安装, 则建议采用国内的 pip 源, 这里推荐采用豆瓣源或者阿里云源进行安装。安装源的配置只需更改 pip.conf 中的 [global] 配置段即可, 具体如下:

```
//豆瓣源配置
[global]
trusted-host = pypi.douban.com
index-url = http://pypi.douban.com/simple
//阿里云源配置
[global]
trusted-host=mirrors.aliyun.com
index-url=http://mirrors.aliyun.com/pypi/simple/
```

pip 在安装过程中, 安装文件默认缓存在 \$HOME/.cache/pip 目录, 在该目录中会生产很多以阿拉伯数字或字母命名的子目录, 这些子目录中存放着 pip 安装时下载的缓存文件。pip 缓存目录如下:

```
[root@mitaka http]# pwd
/root/.cache/pip/http
[root@mitaka http]# ll
total 0
drwx-----.  9 root root 62 Jul 27 16:22 0
drwx-----.  6 root root 38 Jul 27 16:23 1
drwx-----. 13 root root 94 Jul 27 16:23 2
drwx-----. 11 root root 78 Jul 27 16:22 3
drwx-----. 13 root root 94 Jul 27 16:22 4
drwx-----. 10 root root 70 Jul 27 16:23 5
drwx-----. 12 root root 86 Jul 27 16:22 6
drwx-----. 11 root root 78 Jul 27 16:22 7
drwx-----.  9 root root 62 Jul 27 16:22 8
drwx-----. 13 root root 94 Jul 27 16:23 9
drwx-----. 12 root root 86 Jul 27 16:21 a
drwx-----. 13 root root 94 Jul 27 16:23 b
drwx-----. 11 root root 78 Jul 27 16:23 c
drwx-----. 11 root root 78 Jul 27 16:23 d
drwx-----.  9 root root 62 Jul 27 16:23 e
drwx-----. 11 root root 78 Jul 27 16:17 f
```

由于 pip 缓存了安装文件, 因此用户在 uninstall 后, 如果想要重新安装, 则可以直接使用缓存中的软件包进行安装, 而不必要再接入互联网重新下载安装包。使用本地缓存进行安装的命令如下:

```
pip install package_name --src $HOME/.cache
```

或者如下:

```
pip install -r requirements.txt --src $HOME/.cache
```

这里的 requirements.txt 文件是多个需要安装的软件包的集合，pip 会根据 requirements.txt 文件中的软件包名称和版本号自动进行下载安装，这为 pip 进行批量安装提供了一种实现方式。实际上，在进行 OpenStack 的源代码安装时，OpenStack 每个项目都会提供一个 requirements.txt 文件，这个文件里记录了该服务项目所需要的 Python 依赖包。在对 OpenStack 项目进行源码安装（python setup.py install）之前，必须先安装 requirements.txt 文件中所要求的包。如下便是 OpenStack Mitaka 版本 Nova 项目（Nova-13.1.1）提供的 requirements.txt 文件的内容：

```
# The order of packages is significant, because pip processes them in the
# order of appearance. Changing the order has an impact on the overall
# integration process, which may cause wedges in the gate later.
//下述即是Nova所需的全部依赖包，安装Nova前必须先安装下述依赖包
pbr>=1.6 # Apache-2.0
SQLAlchemy<1.1.0,>=1.0.10 # MIT
boto>=2.32.1 # MIT
decorator>=3.4.0 # BSD
eventlet!=0.18.3,>=0.18.2 # MIT
Jinja2>=2.8 # BSD License (3 clause)
keystonemiddleware!=4.1.0,>=4.0.0 # Apache-2.0
.....
```

直接对 requirements.txt 包含的依赖包进行安装的命令如下：

```
pip install -r requirements.txt
```

如果 requirements.txt 文件指定的软件包已经全部下载到本地文件系统中，则可以通过离线方式进行依赖包的安装，安装方式如下：

```
//指定包文件路径进行离线安装
pip install -r requirements.txt -d /path/to/packages
//取消索引，以本地文件为pip源进行离线安装
pip install -r requirements.txt --no-index -f file:///path/to/packages
```

对于网络环境较差或者大规模部署的场景，离线部署是最佳选择，用户仅需下载一次软件包，即可进行多次快速安装部署。此外，如果将已下载的软件包通过 NFS 等网络共享文件系统共享到集群节点，则集群节点可以同时进行离线批量安装。在进行 pip 离线安装之前，通常需要准备 pip 本地源。pip 本地源可以通过两种方式来实现，首先利用 pip2pi 工具下载软件依赖包，然后利用 Apache 配置 HTTP 服务以提供软件包的本地 HTTP 下载，或者通过配置 pypiserver 服务来监听 pip 安装请求并提供本地软件包下载服务。下面对这两种 pip 本地源的制作方式进行介绍。

1. pip2pi 结合 Apache

此方法的主要思想是利用 pip2pi 工具下载 Python 包，之后配置 Apache 服务器以提供本地软件包的 HTTP 下载。基于 pip2pi 和 Apache 的 pip 本地源配置步骤如下：

1) 创建软件包存放目录:

```
mkdir -p /root/pipy
```

2) 安装 pip2pi, 下载所需软件包。pip2pi 安装命令如下:

```
pip install pip2pi
```

3) 下载软件包。软件包批量下载有多种方式, 可以直接使用 pip 工具编写 shell 脚本下载, 也可以通过上一步安装的 pip2pi 直接批量下载。如果通过 pip 进行批量下载, 则可以将集群部署所需的全部 requirements.txt 合并到相同文件中, 具体如下:

```
find / -name requirements.txt -exec cat {} \;> pip_requirements_all.txt
```

将全部依赖软件包整合进一个依赖文件后, 利用循环语句进行批量下载, 脚本语句可参考如下代码:

```
#!/bin/bash
pip_require="pip_requirements_all.txt"
while read LINE
do
    if [[ $LINE =~ ^[a-zA-Z] ]]
    then
        echo $LINE
        pip install $LINE -d /root/pypi //仅下载不安装, 建议使用国内源下载
    fi
done < $ pip_require
```

当然, 如果使用 pip2pi 工具下载, 则无须编写脚本, 仅使用命令行即可进行批量下载。使用 pip2pi 工具下载单个依赖包, 命令如下:

```
pip2tgz /root/pypi pypi_name
```

使用 pip2pi 工具批量下载软件包, 命令如下:

```
pip2tgz /root/pypi -r pip_requirements_all.txt
```

采用 pip2pi 工具提供的 pip2tgz 命令行会自动将 pip_requirements_all.txt 依赖文件中指定的软件包全部下载到指定的目录 (/root/pypi) 中。为了便于后续配置 Apache 服务器, 软件包下载存放目录建议设置为 /var/www/html/pypi, 例如要下载 keystone 项目所需的全部依赖包, 可以通过如下命令实现:

```
pip2tgz /var/www/html/pypi -r /openstack/keystone/requirements.txt
```

由于 pip2pi 默认使用 pypi.python.org 提供的源进行下载, 为了提高下载速度, 建议配置前文所述的国内豆瓣源或阿里源进行下载。

4) 生成软件包索引。软件包下载到本地文件系统后, 需要为全部软件包生成索引 (Index), 这样 pip 在安装查询时可以快速判断指定的依赖软件包是否存在于本地 pip 源中。索引可以使用 dir2pi 命令快速生成, 命令如下:

```
dir2pi --normalize-package-names /var/www/html/pypi
```

dir2pi 命令将会在 /var/www/html/pypi 目录生成 simple 子目录，simple 文件中是下载到本地的全部软件包标准化后的简称。simple 目录中的内容如下：

```
[root@ mitaka pypi]# cd /var/www/html/pypi/simple
[root@ mitaka simple]# ls -l
total 8
drwxr-xr-x. 2 root root 62 Jul 28 17:53 aioeventlet
drwxr-xr-x. 2 root root 50 Jul 28 17:53 alembic
drwxr-xr-x. 2 root root 57 Jul 28 17:53 amqp
drwxr-xr-x. 2 root root 50 Jul 28 17:53 anyjson
drwxr-xr-x. 2 root root 64 Jul 28 17:53 appdirs
```

每个软件包在 simple 目录中都会生成对应子目录，目录名称为标准化后的软件包名。simple 中每个以软件包名称命名的子目录下都会生成一个 index.html 文件，如 WebOb 软件包对应 simple 目录中的子目录为 WebOb，而 WebOb 子目录中 index.html 文件的内容如下：

```
[root@ mitaka simple]# cd WebOb
[root@ mitaka WebOb]# more index.html
<a href='WebOb-1.4.1.tar.gz'>WebOb-1.4.1.tar.gz</a><br />
```

5) 配置 Apache 服务器。确认已经安装 httpd 软件包，在 /etc/httpd/conf.d 目录中新创建 httpd 虚拟机配置文件 pip-server.conf。虚拟机配置文件内容如下：

```
<VirtualHost *:80>
ServerAdmin ynwssjx@126.com
ServerName pip.warrior.com
DocumentRoot /var/www/html/pypi
<Directory /var/www/html/pypi>
Options Indexes FollowSymLinks MultiViews
Allow Override None
Require all granted
</Directory>
ErrorLog logs/mirrors-error_log
CustomLog logs/mirrors-access_log common
</VirtualHost>
```

Apache 服务器配置完成后，重新启动 httpd 服务：

```
systemctl restart httpd.service
netstat -ntln |grep 80
```

6) 验证软件包 HTTP 下载。在浏览器输入 http://host_ip/pypi/simple，页面将以超链接的形式显示全部可供本地下载的软件包名称，如图 11-1 所示。此时，单击任意软件包名称，浏览器便会自动下载对应的软件包。

7) 使用本地 pip 源安装软件。截至步骤 6，pip 本地源已经配置完成，现在即可使用 pip 本地源进行依赖包安装。用户可以通过 pip 安装命令行指定本地源路径，命令如下：

```
pip install --index-url http://host_ip/pypi/simple -r requirements.txt
```

也可以通过修改 `pip.conf` 配置文件一劳永逸地修改 `pip` 安装的默认源，使其默认使用本地 `pip` 源安装软件，修改方式如下：

```
[global]
//////////douban mirrors//////////
#trusted-host=pypi.douban.com
#index-url = http://pypi.douban.com/simple
//////////pip2pi&apache local repos//////////
trusted-host=192.168.142.10
index-url = http://192.168.142.10/pypi/simple
```

这样，用户便可使用如下命令进行批量依赖包的离线安装：

```
pip install -r requirements.txt
```

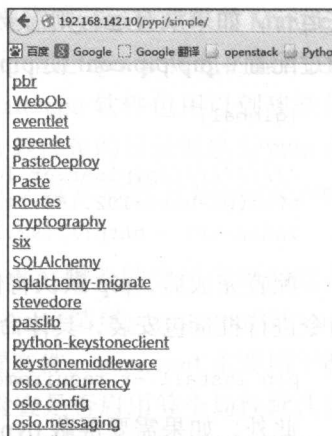


图 11-1 pip 本地源 HTTP 下载页面

通过这种方式，用户可以在集群中的多个节点上指定同一个本地 `pip` 源，并通过 HTTP 协议从本地 Apache 服务器上下载安装软件包。比起从 `www.pypi.org` 或者其他 `pip` 源下载进行安装，`pip` 本地源的安装方式要快得多。

2. pip2pi 结合 pypiserver

利用 `pip2pi` 和 Apache 服务器结合的方式可以很好地提供 `pip` 本地源服务，但是配置 Apache 服务器相对复杂且不易理解，为了简单起见，用户也可以通过 `pip2pi` 与 `pypiserver` 的组合方式来提供本地 `pip` 源服务。这种方式通常只需两个步骤即可实现 `pip` 本地源：第一步是利用 `pip2pi` 将依赖包的批量下载到特定目录；第二步是启动 `pypiserver` 服务（需要预先通过 `pip install pypiserver` 命令安装 `pypiserver`），并在启动过程中指定服务监听端口和软件包的存放路径。现在假设软件包已经下载到本地 `/root/pypi` 目录中，则可以通过如下命令启动 `pypiserver` 服务进程：

```
pypi-server -i host_ip -p port packages_dir1 packages_dir2 ... &
```

其中，`host_ip` 为启动 `pypiserver` 服务的主机，`port` 为 `pypiserver` 服务监听端口，用户可以同时指定多个本地软件包路径。`pypi-server` 命令行工具不需要使用 `dir2pi` 生成软件包 Index，只需将需要的 python 包下载到本地，然后启动 `pypiserver` 监听服务即可。`pypi-server` 可以使用 `--fall-back` 参数设置在指定监听的包目录中找不到需要的软件包时，`pip` 的候选 `pip` 源名称。如下命令将前面配置的 HTTP 源作为 `pip` 的候选源：

```
pypi-server -i 192.168.142.10 -p 8080 --fallback-url \
http://192.168.142.10/pypi/simple /root/pypi &
```

上述启动命令中，`pip` 首先在 `/root/pypi` 目录进行软件包搜索，如果没有找到，则转入 `http://192.168.142.10/pypi/simple` 这个 HTTP 服务器中搜索。此外，`pypiserver` 服务进程必

须以后台进程形式启动,在使用 pip 进行本地安装之前,建议先检查 pypiserver 进程是否正在运行,如果没有运行则需要以后台进程形式重新启动。pypiserver 进程启动完成后,可以通过配置 `~.pip/pip.conf` 使 pip 默认使用本地 pip 源进行软件包安装,配置方式如下:

```
[global]
.....
//////////pypiserver local repos//////////
trusted-host=192.168.142.10
index-url = http://192.168.142.10:8080/simple/
```

配置完成后, pip 默认将使用本地源进行包安装。此时,用户便可使用常规的 pip 安装命令进行批量包安装,具体命令如下:

```
pip install -r requirements.txt
```

此外,如果需要屏蔽 pypiserver 提供的本地 pip 源服务,则只需停止 pypiserver 服务进程即可。由于 pypiserver 运行在后台,通常可以使用 `lsuf` 命令查出 pypiserver 的 PID,并通过 `kill` 命令将其停止即可。

11.1.2 制作 OpenStack 离线安装 yum 源

在 Linux 系统的软件包管理工具中, rpm 和 yum 是两个最常见的工具。rpm 是由 Redhat 公司开发的包管理工具,即 Redhat Package Management 的缩写。rpm 可用于软件包的安装、卸载和查看等操作,但是 rpm 包并不能解决软件包依赖问题。而 Linux 作为一个开源系统平台,很多软件包的安装和运行需要依赖额外软件,如在使用 rpm 命令安装某个软件包时,可能需要先安装另一个依赖软件包,而 rpm 并不能自动处理这种依赖关系。相反, yum 软件包管理工具却可以自动处理软件包之间的依赖关系,并能够通过网络 yum 源自动进行依赖查询和安装。因此, yum 是 Linux 系统中非常重要的软件包管理工具。在 OpenStack 的离线部署中,用户可以事先同步网络 yum 源至本地,并将同步到本地的软件包制作成本地 yum 源,之后利用本地 yum 进行 OpenStack 集群节点的快速安装部署。

1. yum 基础介绍

yum 是 Yellowdog Updater Modified 的简称。yellowdog 是一个 Linux 发行版。yum 最初便是由 yellowdog 发行版的开发者 Terra Soft 利用 Python 语言开发而来的,当然, Terra Soft 开发时还称之为 yup (yellow dog updater)。之后杜克大学的 Linux 开发团队对其进行了改进,之后便将其称之为 yum (Yellowdog Updater Modified)。yum 的主要任务就是自动处理与软件包相关的操作,如软件升级、安装或移除 rpm 包、收集 rpm 包的相关信息以及检查软件包依赖性并自动将依赖关系告知用户。yum 最重要的地方在于配置 yum 仓库 (Repository),即 yum 源。Repository 可以是 HTTP 或 FTP 站点,也可以是本地目录或文件夹 (File)。对于任何一个 yum 仓库,都必须生成一张包含全部软件元数据信息的清单表

(Manifest), 元数据信息又称 rpm 软件包的 header, header 包括了 rpm 包的各种信息, 如软件包的描述、功能介绍、包含的文件以及依赖的软件包列表等。yum 仓库中的 Manifest 对全部 rpm 软件进行了汇总统计, 当用户在使用 yum 命令进行指定 rpm 软件包的操作时, yum 便会查询 Manifest 以获取相关的信息。Linux 提供了 Createrepo 软件包用以创建软件仓库, 在制作本地 yum 源时, 通常利用 Createrepo 将包含 rpm 软件包的目录创建为 yum 仓库, 然后在 /etc/yum.repos.d 中配置指向该仓库的 yum 源即可 (通常命名为 filename.repo 形式)。

yum 有两类配置文件: 一个是 yum.conf 配置文件, 这是 yum 服务的通用全局配置文件; 另一个是 yum 特定仓库配置文件, 仓库配置文件通常以 .repo 结尾。yum.conf 是 yum 的全局性配置文件, 位于 /etc 目录, 通常情况无须更改此配置文件。yum.conf 主要用于配置 yum 的缓存目录、运行调式级别、日志文件、插件和 gpg 检查是否启用等全局性默认参数。如下是 Centos7 系统中的 yum.conf 配置文件内容, 不同 Linux 版本的 yum.conf 内容类似:

```
[root@mitaka ~]# more /etc/yum.conf
[main]
cachedir=/var/cache/yum/$basearch/$releasever //yum下载软件包的缓存目录
keepcache=0 //是否保存缓存, 1为保存
debuglevel=2 //调式级别 (0-10)
logfile=/var/log/yum.log //yum日志文件
exactarch=1 //是否允许更新不同架构的rpm包, 如是否在i386上更新i686的rpm包
obsoletes=1 //是否允许update陈旧的rpm包, 1为允许
gpgcheck=1 //是否进行gpg (GNU Private Guard) 检查
plugins=1 //是否允许使用插件, 1为允许, 通常需要使用yum-fastestmirror插件
installonly_limit=5 //允许保留多少kernel包
bugtracker_url=http://bugs.centos.org/set_project.php?project_id=23&ref=http://
    bugs.centos.org/bug_report_page.php?category=yum
distroverpkg=centos-release //告知yum系统版本
.....
# metadata_expire=90m //rpm包元数据到期时间
# PUT YOUR REPOS HERE OR IN separate files named file.repo in
# /etc/yum.repos.d
```

yum 特定仓库配置文件位于 /etc/yum.repos.d 目录, 通常特定的 yum 仓库对应一个独立的仓库配置文件 (也可以在同一个配置文件中指定多个仓库源)。对于 Centos 7 系统, 默认的 yum 仓库配置文件如下:

```
[root@mitaka ~]# ls -l /etc/yum.repos.d
total 28
-rw-r--r--. 1 root root 1664 Dec 9 2015 CentOS-Base.repo
-rw-r--r--. 1 root root 1309 Dec 9 2015 CentOS-CR.repo
-rw-r--r--. 1 root root 649 Dec 9 2015 CentOS-Debuginfo.repo
-rw-r--r--. 1 root root 290 Dec 9 2015 CentOS-fasttrack.repo
-rw-r--r--. 1 root root 630 Dec 9 2015 CentOS-Media.repo
```



```
-rw-r--r--. 1 root root 1331 Dec 9 2015 CentOS-Sources.repo
-rw-r--r--. 1 root root 1952 Dec 9 2015 CentOS-Vault.repo
```

其中，每一个 repo 文件都可以包含一个或多个 yum 源仓库。以 CentOS-Base.repo 为例，该仓库配置文件中的内容如下：

```
[root@mitaka bak]# more CentOS-Base.repo
.....
[base]
name=CentOS-$releasever - Base
mirrorlist=http://mirrorlist.centos.org/?release=$releasever&arch=\
$basearch&repo=os&infra=$infra
#baseurl=http://mirror.centos.org/centos/$releasever/os/$basearch/\
gpgcheck=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-7
.....
```

CentOS-Base.repo 中一共配置了 base、updates、extras 和 centosplus 四个 yum 源，每个 yum 源的配置形式都是固定的，通常由 name、mirrorlist（或 baseurl）、gpgcheck、enabled 和 gpgkey 几个参数组成。其中，name 是由用户自定义的 yum 源名称，可以根据 yum 源的功能特性进行自定义命名。mirrorlist 或 baseurl 用于指定 yum 仓库源的位置，baseurl 的值是一个具体的 rpm 软件包仓库，该仓库中通常包含一个存放 rpm 包元数据信息的 repodata 目录，而 mirrorlist 是一个存放 rpm 包仓库链接的列表网站，yum 会在 mirrorlist 指定的列表中选择合适的 rpm 仓库作为软件下载或更新的 yum 源。enabled 参数用于设置是否启用该 yum 源，仅有 enabled 为 1 的情况下，yum 才会搜索此 yum 源。gpgcheck 和 gpgkey 主要用于设置 yum 仓库的 gpg 校验，gpgcheck 参数用于设置是否需要进行 gpg 验证，如果值为 0 则不需要，此时也无须设置 gpgkey 参数；如果值为 1 则表示需要进行验证，则此时需要设置 gpgkey 以提供 gpg 验证码。

2. 同步 OpenStack 安装包至本地

此处以制作基于 Redhat 发行的 OpenStack 版本（RDO，Redhat Distribution OpenStack）本地 yum 源为例，进行 OpenStack 离线安装 yum 源制作的讲解。根据不同的 OpenStack 版本，用户可能需要通过不同的方式启用 OpenStack repository^①，这里以在 CentOS 7 系统中部署 Mitaka 版本为例。为了支持 RDO 的 OpenStack repository，通常需要启用 extras repository，而 Centos 7 的 CentOS-Base.repo 中默认已经存在 extras repository，因此仅需在 Centos 7 系统中安装 OpenStack 的 Mitaka 版本 rpm 包仓库即可：

```
yum install centos-release-openstack-mitaka
```

安装完成之后，/etc/yum.repos.d 中将会增加几个与 OpenStack 和 Ceph 相关的 .repo 配置文件，这些配置文件中已经设置了 OpenStack 和 Ceph 的 yum 源仓库。这几个新增的

① 具体方式参见 [http://docs.openstack.org/\\$release/install-guide-rdo/environment-packages.html](http://docs.openstack.org/$release/install-guide-rdo/environment-packages.html)。

repository 配置文件分别是 CentOS-OpenStack-mitaka.repo、CentOS-Ceph-Hammer.repo 和 CentOS-QEMU-EV.repo。/etc/yum.repos.d 目录中的 repository 配置文件如下：

```
[root@mitaka yum.repos.d]# ls -l /etc/yum.repos.d
total 40
-rw-r--r--. 1 root root 1664 Dec  9 2015 CentOS-Base.repo
-rw-r--r--. 1 root root 1057 Feb 25 2016 CentOS-Ceph-Hammer.repo
-rw-r--r--. 1 root root 1309 Dec  9 2015 CentOS-CR.repo
-rw-r--r--. 1 root root  649 Dec  9 2015 CentOS-Debuginfo.repo
-rw-r--r--. 1 root root  290 Dec  9 2015 CentOS-fasttrack.repo
-rw-r--r--. 1 root root  630 Dec  9 2015 CentOS-Media.repo
-rw-r--r--. 1 root root 1128 Oct  6 03:30 CentOS-OpenStack-mitaka.repo
-rw-r--r--. 1 root root  578 Oct  6 2015 CentOS-QEMU-EV.repo
-rw-r--r--. 1 root root 1331 Dec  9 2015 CentOS-Sources.repo
-rw-r--r--. 1 root root 1952 Dec  9 2015 CentOS-Vault.repo
```

其中，CentOS-OpenStack-mitaka.repo 中设置了 Mitaka 版本 OpenStack 的 rpm 安装包 yum 源，其内容如下：

```
[root@mitaka yum.repos.d]# more CentOS-OpenStack-mitaka.repo
.....
[centos-openstack-mitaka]
name=CentOS-7-OpenStack mitaka
baseurl=http://mirror.centos.org/centos/7/cloud/$basearch/openstack-mitaka
gpgcheck=1
enabled=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-SIG-Cloud
.....
```

在 CentOS-OpenStack-mitaka.repo 中，仅启用了 centos-openstack-mitaka 这个 yum 源，该 yum 源中指定了安装 OpenStack 所需 rpm 包的仓库路径。正常情况下，此时已经可以进行在线安装 OpenStack，而如果需要进行离线安装，则还需到 centos-openstack-mitaka 这个 yum 源设置的 rpm 仓库中下载 Redhat 封装的 OpenStack Mitaka 版本 rpm 软件包到本地。下载 yum 仓库中的 rpm 包有两种方式：一种是通过 rsync 命令远程同步 rpm 包到本地；另一种是通过 reposync 自动以 repository 为单位同步 rpm 包到本地。rsync 是一个功能相当丰富的文件同步工具，可以基于 TCP/IP 网络将远程文件同步到本地，并且具有丰富的参数可供选择。如果采用 rsync 进行 OpenStack rpm 安装包的同步，则可以参考如下 OpenStack 安装包同步脚本：

```
#!/bin/bash
#This Script used to download openstack mitaka rpm packages to local
rsyncBin="/usr/bin/rsync" //命令路径，如果没有则需要安装rsync包
rsyncPerm='-avrt --delete --no-iconv --bwlimit=1000' //rsync执行参数
Local_path='/data/openstack-mitaka' //本地存放路径
LogFile='/data/yum_repo/rsync_yum_log' //存放同步操作的日志路径
Date=`date +%Y-%m-%d` //日期设置
#check
```

```

function check_rsync()
{
if [ $? -eq 0 ];then
    echo "Rsync is success!" >>$LogFile/$Date.log
else
    echo "Rsync is failed! " >>$LogFile/$Date.log
fi
}
if [ ! -d "$LogFile" ];then
    mkdir -p $LogFile
fi
if [ ! -d "$Local_path" ];then
    mkdir -p $Local_path
fi
//开始同步openstack-mitaka
echo 'Now start to rsync openstack-mitaka!' >>$LogFile/$Date.log
$RsyncBin $RsyncPerm rsync://mirror.centos.org/centos/7/cloud/x86_64\
/openstack-mitaka $Local_path >>$LogFile/$Date.log 2>&1
//同步完成后验证同步是否成功
check_rsync

```

相比之下，由 yum-utils 提供的 reposync yum 源同步工具是更为方便的选择，reposync 的语法如下：

```
reposync --repoid=REPOID --download_path=DESTDIR
```

其中，--repoid 用于指定需要同步到本地的 yum 仓库 ID；--download_path 用于指定存放 rpm 包的本地路径。reposync 可以根据用户指定的 repo ID 进行同步，可以通过 yum 提供的 repolist 命令查看 repo ID，具体如下：

```
[root@mitaka openstack-mitaka]# yum repolist
.....

```

repo id	repo name	status
base/7/x86_64	CentOS-7 - Base	9,007
centos-ceph-hammer/7/x86_64	CentOS-7 - Ceph Hammer	40
centos-openstack-mitaka/x86_64	CentOS-7 - OpenStack mitaka	1,582
centos-qemu-ev/7/x86_64	CentOS-7 - QEMU EV	52
extras/7/x86_64	CentOS-7 - Extras	393
updates/7/x86_64	CentOS-7 - Updates	2,560
repolist: 13,634		

repolist 的输出中有三列，其中第一列便是与各个 yum 仓库对应的 ID。这里，OpenStack Mitaka 的 yum 仓库对应的 ID 为 centos-openstack-mitaka/x86_64。因此，要同步基于 Mitaka 的 RDO 安装包到本地 /data/openstack-mitaka 目录，则同步命令如下：

```

reposync --repoid=centos-openstack-mitaka/x86_64\
--download_path=/data/openstack-mitaka

```

同步所需时间取决于仓库大小和网络速度，就 OpenStack Mitaka RDO 源而言，同步完


```
[root@mitaka ~]# createrepo --baseurl=/data/openstack-mitaka \
/data/openstack-mitaka
.....
```

创建成功之后，在 /data/openstack-mitaka 目录下将会看到 repodata 目录：

```
[root@mitaka openstack-mitaka]# ls -ld repodata
drwxr-xr-x 2 root root 4096 May 18 11:37 repodata
```

repodata 目录中存放了与此 yum 仓库相关的元数据信息，yum 通过这些元数据与仓库进行交互。如果当前目录中已经存在 repodata 目录，而仅希望更新当前 yum 仓库，则可以使用 createrepo 的 --update 参数，命令如下：

```
[root@mitaka ~]# createrepo --baseurl=/data/openstack-mitaka --update \
/data/openstack-mitaka
```

yum 仓库创建完成后，需要在 /etc/yum.repos.d 中进行相应的配置，以便 yum 命令能够直接访问创建的 yum 仓库。本例中，在 /etc/yum.repos.d 中创建一个 repository 配置文件 openstack-mitaka.repo，其内容如下：

```
[root@mitaka yum.repos.d]# more openstack-mitaka.repo
[openstack-mitaka]
name=openstack-mitaka                //自定义仓库名称
baseurl=file:///data/openstack-mitaka //仓库地址
enabled=1                             //启用此仓库
gpgcheck=0                            //不进行GPG验证
```

配置完成后，通常需要清除 yum 中原有的 cache，并重新生成 cache。清除 yum 陈旧 cache 的命令如下：

```
[root@mitaka yum.repos.d]# yum clean all
```

然后，重新生成 cache，命令如下：

```
[root@mitaka yum.repos.d]# yum makecache
```

至此，已经可以使用本地 yum 源进行离线 OpenStack 安装。下面以安装 OpenStack 对象存储 Swift 为例。这里安装与 Swift 相关的全部软件包，正常情况下 yum 应该自动解析依赖并到本地 openstack-mitaka 仓库中下载 Swfit 软件包和依赖包并进行安装，安装过程如下：

```
[root@mitaka ~]# yum install openstack-swift-*
.....
Install 7 Packages (+3 Dependent packages)
Total download size: 2.0 M
Installed size: 8.4 M
Is this ok [y/d/N]:
```

从上述安装过程中可以看到，yum 自动解析与 Swift 相关的软件包，并在 openstack-mitaka 这个本地 yum 仓库中查询到了全部所需的安装包，只要用户确认安装并输入“y”，

yum 就会自动到 /data/openstack-mitaka 目录中提取 RPM 安装包进行安装。与在线网络安装相比, 基于本地 yum 源的离线安装速度要快得多, 并且通过 NFS 共享方式, 可以向多个节点发送并行命令进行批量安装。尤其是在大规模集群安装部署中, 制作 OpenStack 本地 yum 源并进行离线安装是非常必要的。

11.2 OpenStack 集群高可用部署架构设计

对于 OpenStack 高可用集群部署而言, 笔者建议在生产环境中正式部署之前, 先在实验环境中进行 OpenStack 各个功能模块和高可用性的验证。通过实验环境从理论上完全实现了前期需求分析中对 IaaS 基础架构的各种需求之后, 生产环境中的部署将会变得非常快速和简单 (通常只需将实验环境中部署成功的代码复制一份, 并放到生产环境中运行即可), 而且不会因为很多意外问题而额外占用很多诸如电力、网络等生产环境资源。因此, 本节将从实验环境准备和生产环境准备两个方面来介绍 OpenStack 高可用集群部署的硬件环境搭建。值得指出的是, 由于 OpenStack 部署模式的灵活性, 事实上并不存在绝对唯一的部署架构, 例如不同的服务可以合并部署到一个节点, 也可以分开部署到不同节点, 而多个服务又可以以不同的组合形式部署。因此, 这里给出的仅是参考实现架构, 不同用户根据自身的实际环境, 通常需要作出适当的调整。

11.2.1 OpenStack 高可用部署实验环境架构

为了实现一个“移动式” OpenStack 高可用实验环境, 笔者建议将实验环境部署在一台笔记本电脑上, 以方便随时随地全身心投入 OpenStack 的研究部署中, 当然也可以部署在数据中心 PC Server 上^①。由于普通笔记本物理资源, 尤其是内存, 难以满足集群需求, 因此在开始 OpenStack 环境搭建之前, 可能需要对个人笔记本进行硬件升级。这里硬件升级通常指升级内存条。不同型号笔记本支持的最大内存受主板限制可能不一样, 较老型号的可能仅支持最大 4GB 或 8GB, 而较新的主板可能支持 16GB 或 32GB。查看主板支持的最大内存方式如图 11-3 所示, 在 cmd 命令窗口中输入 wmic memphysical get maxcapacity 命令即可查看, 图 11-3 中笔记本最大支持 16GB 内存。

由于 OpenStack 高可用集群实验环境仅做功能验证测试, 不做性能压力测试, 因此根据笔者经验, 16GB 内存通过虚拟化完全可以实现三个控制节点和两个计算节点的 OpenStack 高可用集群实验环境。此外, 如果希望在此电脑上进行 Ceph 存储使用, 则建议增大硬盘容量, 1TB 的硬盘空间足以满足存储节点实验和虚拟机的多次快照保存。以笔者的实验环境为例, 笔记本型号为联想 T440p, 内存 16GB, CPU 为 Intel 酷睿 i5 处理器, 存储为两块 500GB 硬盘, 操作系统为 64 位 Windows7 SP1 旗舰版, 虚拟化软件为 VMware

① 将实验环境搭建在 Laptop 上是强烈推荐的方式, 数据中心的实验环境通常在使用上受到很多限制。

Workstaion11, 一共创建 5 台 VMware 虚拟机, 其中 3 台控制节点用于实现 OpenStack 服务的高可用, 2 台计算节点用于实现实例 HA。另外, 由于 OpenStack 高可用环境中必须实现 Quorum 机制, 并能够对控制节点进行隔离操作 (Fencing), 而 VMware 虚拟机在实现 Fencing 上有一定困难, 因此在三台控制节点 VMware 虚拟机上又各自创建了一台 KVM 虚拟机, 并且将这三台 KVM 虚拟机作为最终的 OpenStack 高可用集群控制节点。笔者的实验环境物理架构如图 11-4 所示。在图 11-4 中, 控制节点由三台 VMware 虚拟机中的三台 KVM 虚拟机组成。计算节点由两台 VMware 虚拟机组成, OpenStack 高可用集群由运行 Pacemaker 和 Corosync 进程的控制节点, 以及运行 Pacemaker_remote 的计算节点组成。计算节点之所以仅运行 Pacemaker_remote, 是因为生产环境中如果计算节点也运行 Pacemaker 和 Corosync 进程, 则 Pacemaker 高可用集群最大仅支持 16 个节点, 而这显然不能满足生产环境中大规模节点部署的需求。Pacemaker_remote 为 Redhat 专为解决 Pacemaker 集群节点限制而开发的精简版 Pacemaker, 计算节点在运行 Pacemaker_remote 时, 并不存在最大 16 个节点的限制, 并且运行 Pacemaker_remote 的计算节点完全可以作为 Pacemaker 集群节点存在并接受控制节点的控制。

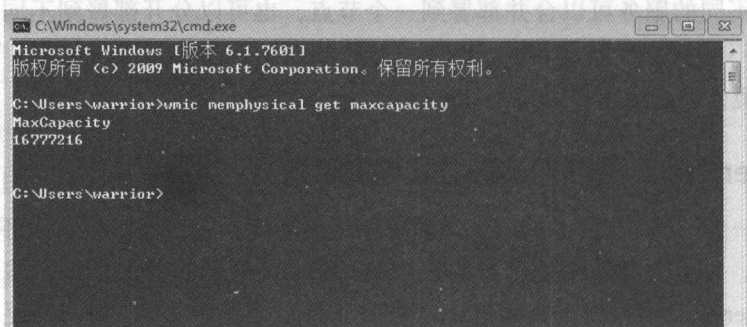


图 11-3 查看主板支持最大内存

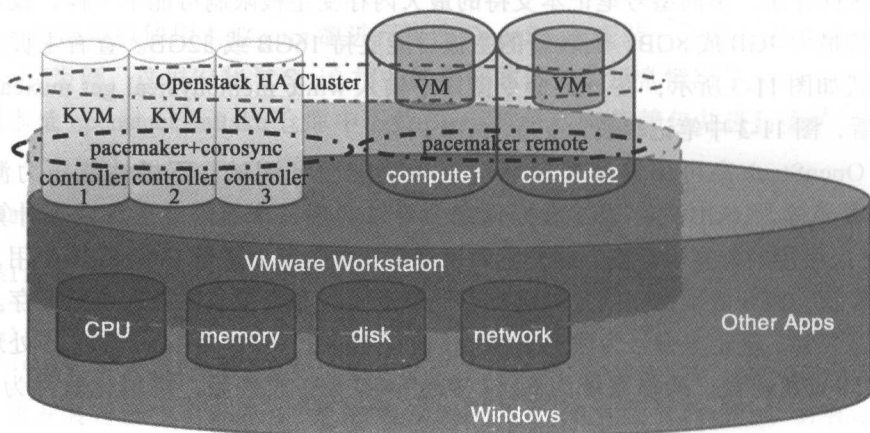


图 11-4 OpenStack 高可用部署实验环境架构

在基于图 11-4 所示实现的实验环境中，服务器节点的部署实现所需要准备的工作便是准备好 5 台 VMware 虚拟机（3 台为控制节点，2 台为计算节点）。控制节点 VMware 虚拟机的配置如图 11-5 所示。

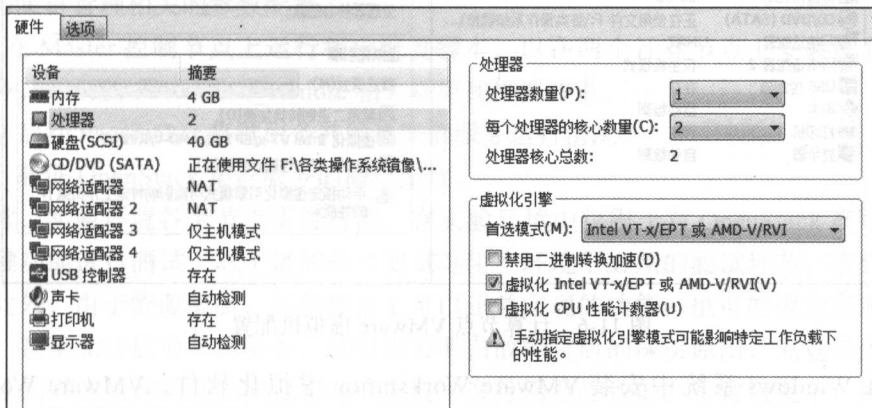


图 11-5 控制节点 VMware 虚拟机配置

为了能够在 VMware 虚拟机中嵌套创建 KVM 虚拟机，在 VMware 虚拟机设置的处理器选项中，为处理器选择 Intel VT-x/EPT 或 AMD-V/RVI 虚拟化引擎。此外，处理器的数量不易设置过大，否则容易出现因资源分配问题而导致的 VMware 虚拟机 Holding 问题。如果对 VMware 虚拟机 CPU 数量有要求，可以通过每个处理器的核心数来控制。由于 3 台 VMware 虚拟机需要嵌套创建 KVM 虚拟机来作为控制节点，并且 3 台 KVM 虚拟机控制节点需要运行绝大部分的 OpenStack 相关基础服务和 OpenStack 核心服务，因此建议分配相对较多的内存。图 11-5 所示分配了 4GB 内存给 VMware 虚拟机，其中的 3GB 用于 VMware 虚拟机中创建的 KVM 虚拟机控制节点。与控制节点不同，计算节点主要负责用户实例创建，如果资源允许，计算节点可以分配较多的 CPU 和内存等资源。此外，对于一个高可用的 OpenStack 集群而言，任何节点都应该支持 Fencing 功能。由于是实验环境，图 11-4 所示的架构中仅运行在 VMware 虚拟机中的 3 个 KVM 虚拟机控制节点通过虚拟机 Fencing 驱动实现了隔离 Fencing 功能，而两个运行在 VMware 虚拟机上的计算节点并没有实现 Fencing，但是这并不影响实验环境中 OpenStack 集群的多数高可用功能验证。如果在生产环境中，则计算节点的 Fencing 功能是必须的。

根据图 11-4 所示的实验环境架构，如果想要在自己的笔记本上部署一套高可用的 OpenStack 实验环境，则可以按照如下步骤进行实现。

步骤 1) 准备一台硬件相对高配的笔记本电脑，进入主板 BIOS，打开虚拟化支持功能（Intel-VT 或 AMD-V）。通常笔记本或 PC 服务器出厂时虚拟化支持功能均会打开，但是仍然建议进行检查，因为后面很多与虚拟化相关的奇怪问题可能均与主板虚拟化功能是否开启有关。

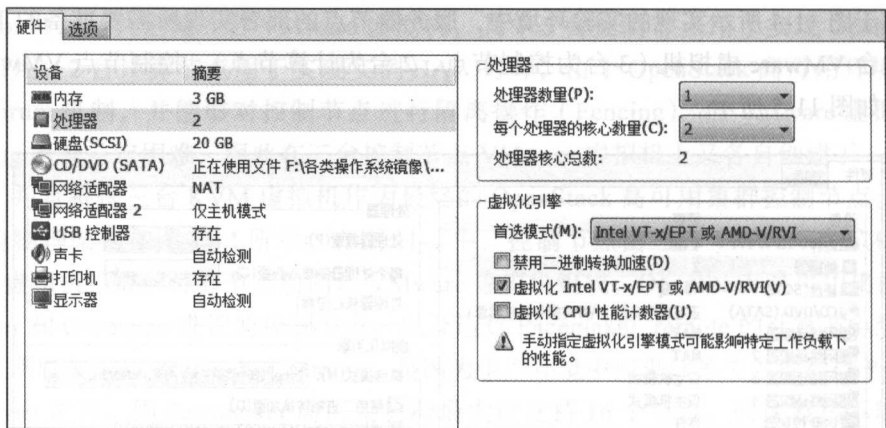


图 11-6 计算节点 VMware 虚拟机配置

2) 在 Windows 系统中安装 VMware Workstation 虚拟化软件, VMware Workstation 与 VMware ESXi 系列虚拟化引擎不一样, Workstation 是一种操作系统层面的虚拟化机制, 其以应用程序的形式运行在操作系统中, 因此在启动实验环境后, 建议不要在同一操作系统里面运行过多的其他应用程序, 尤其是资源密集型的应用程序, 以保证 VMware Workstaion 可以获得最多的资源。

3) 在 VMware Workstation 中创建三台 VMware 虚拟机 (虚拟机名称分别为 Controller1、Controller2 和 Controller3), 虚拟机具体配置可以参考图 11-5 所示。三台 VMware 虚拟机用作 OpenStack 高可用集群控制节点宿主机。VMware 虚拟机安装 Centos7.1 系统。为了最大化 KVM 虚拟机控制节点的资源使用, VMware 虚拟机操作系统安装方式选为最小安装。

4) 在 VMware Workstation 中创建两台 VMware 虚拟机 (虚拟机名称分别为 Compute1 和 Compute2), 两台虚拟机用作 OpenStack 高可用集群的计算节点, 虚拟机具体配置可以参考图 11-6 所示。VMware 虚拟机安装 Centos7.1 系统。为了最大化将计算节点资源应用到用户实例中, 计算节点操作系统选用最小安装方式安装。

5) 将控制节点 Controller1 设置为 Master 控制节点, Master 控制节点将同时充当管理集群节点的角色, 后续此节点将被配置为 NFS 服务器。因为实验环境采用离线安装方式, 在运行部署脚本之前, 先将 Centos71 系统镜像和 OpenStack 离线安装所需的 rpm 软件包和依赖包上传到 Master 上。

6) 运行自动化配置脚本, 初始化 Controller1、Controller2 和 Controller3 这三台 VMware 虚拟机, 为后期部署 OpenStack 服务准备好系统环境。初始化完成后, Controller1 将被配置为 Master 控制节点和集群管理节点, 同时被配置为集群 NFS 服务器。

7) 运行自动化配置脚本初始化 Compute1 和 Compute2 这两台 VMware 虚拟机。初始化完成后, 两台虚拟机将被配置为计算节点环境, 从而为后期部署 OpenStack 的 Nova-

compute 和 Pacemaker_remote 服务准备好系统环境。

8) 将自动化配置所需的全部脚本上传到 Master 控制节点, 并在其上运行相应的脚本以在三个控制节点上安装配置 OpenStack 相关服务, 同时设置 Pacemaker 集群资源和资源约束等与集群管理相关的参数配置。

9) 在 Master 控制节点上运行相应配置脚本, 以在两个计算节点上安装配置 Compute 相关服务, 并设置 Pacemaker_remote 相关资源和集群约束。

10) 启动 Pacemaker 集群服务并验证集群服务运行情况。

11) 验证 OpenStack 集群服务的高可用性。

通常, 实验环境各个节点资源有限, 在实验环境中仅限于对 OpenStack 高可用集群的部分关键功能进行测试, 对于诸如压力测试等生产环境下必须的测试环节, 实验环境无能为力。此外, 由于资源限制, 某些理论上可以正常实现的功能, 也可能因为资源不足而不能实现, 甚至无法启动某些服务。此时需要明白故障背后的深层原因, 究竟是因为资源不足导致还是因为软件问题、配置不对或配置冲突等问题导致的? 正常情况下, 实验环境中可以正常实现的功能, 在迁移至生产环境后, 都可以成功实现, 用户需要注意的是随着集群规模的扩大, 客户端连接数的增加为数据库和消息队列等基础服务带来的影响, 例如数据库的最大连接数设置和消息队列的阻塞等问题。

11.2.2 OpenStack 高可用部署生产环境架构

对于开源软件的使用, 没有任何固定的设计架构, 尤其是像 OpenStack 这类“大帐篷”模式发展的开源社区软件。在云计算不断落地发展的现今, 基于 OpenStack 的创业公司层出不穷, 同时传统 IT 巨头也在借力 OpenStack 加速转型成为云计算公司, 并且很多具有较强 IT 技术实力的企业也在基于 OpenStack 自建私有云, 因此应用于生产环境中的 OpenStack 部署架构可以说是“百花齐放, 百家争鸣”。在 OpenStack 高可用集群架构设计中, 不同厂商均提出了自己的高可用设计架构^①, 这其中以开源领导厂商 RedHat 和 Pure Play OpenStack 厂商 Mirantis 的高可用方案最为主流, 而国内很多 OpenStack 创业公司初期的 OpenStack 高可用方案都源自 RedHat OSP (OpenStack Platform) 和 Mirantis 的 Fuel 系列 OpenStack 部署软件的定制化和二次开发, 因此对于计划自建 OpenStack 高可用集群的用户而言, 如果没有更好的高可用设计架构, 则 RedHat 和 Mirantis 的高可用方案确实是非常值得借鉴的。图 11-7 所示为 RedHat OSP 系列的 OpenStack 高可用集群部署参考架构, 关于 RedHat OSP 高可用架构的具体描述可以参考第 2 章中的相关部分。

从架构的整体设计上来看, RedHat 提供的高可用架构充分考虑了企业级应用服务的高可用性, 因此架构在设计和实现上都需要一定的技术实力和成本预算来支撑, 对于有较多

① 不同厂商的高可用设计方案及其对比可以参考第 2 章。

云计算人才和充分预算的企业，RedHat 方案将是不错的选择。图 11-8 所示是 Mirantis 主导的 OpenStack 高可用架构设计方案之一。Mirantis 是一家完全基于 OpenStack 的初创公司，其设计理论和方案以简单实用为主，在架构设计上不会显得冗余繁重，非常适合对云计算接触并不深入的年轻团队或者在成本预算上受到限制的部门或初创公司进行基于 OpenStack 的云计算建设参考。当然，除了 Redhat 和 Mirantis 的 OpenStack 高可用方案外，用户也可以综合各家所长实现自己的 OpenStack 高可用架构，只要整个架构的最终设计能够满足前期提出的 IaaS 服务需求分析即可。

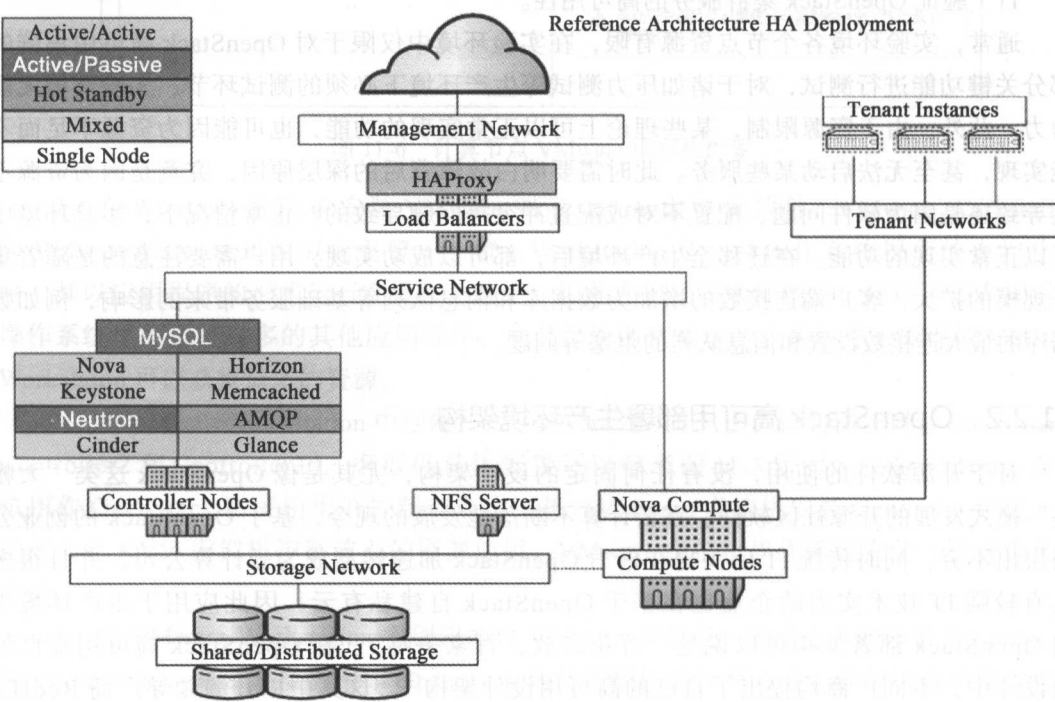


图 11-7 RedHat OSP 高可用集群部署参考架构

对于生产环境而言，硬件服务器节点的拓扑情况通常根据 OpenStack 各个服务组件部署模式的不同选择而不同，由于 OpenStack 服务组件的部署具有极大的灵活性，因此在项目实施之初的设计阶段需要谨慎，以保证 OpenStack 集群在功能正常使用的同时，还应该具备高可靠性、高可用性和水平可扩展性。通常，为了保证生产环境的高可用，控制节点数目至少由 3 个物理服务器构成，而计算节点则根据实际的规划需求可以分为一期上线和后续二期扩容的形式来规划采购。对集群拓扑影响较大的可能是后端存储的采用，如果采用 IBM、EMC、Netapp 或华为等企业存储阵列，则 OpenStack 集群后端存储需要一个复杂的存储 SAN 网络支撑，图 11-9 所示即是比较典型的基于企业级高可用存储的 OpenStack 集群部署架构，OpenStack 集群存储云盘由 Cinder 项目提供，Cinder 后端可以使用 LVM 驱动

或直接使用由特定存储厂商提供的存储驱动来使用底层存储。图 11-9 中，后端底层存储采用 EMC VPLEX 或 IBM SVC 存储异构虚拟化产品对底层不同存储厂商的存储阵列进行异构虚拟化封装，并统一由 VPLEX 或 SVC 作为 Cinder 的后端存储设备，而在 Cinder 的后端存储驱动矩阵中，用户可以轻易找到 EMC 和 IBM 提供的 VPLEX 和 SVC 驱动程序。

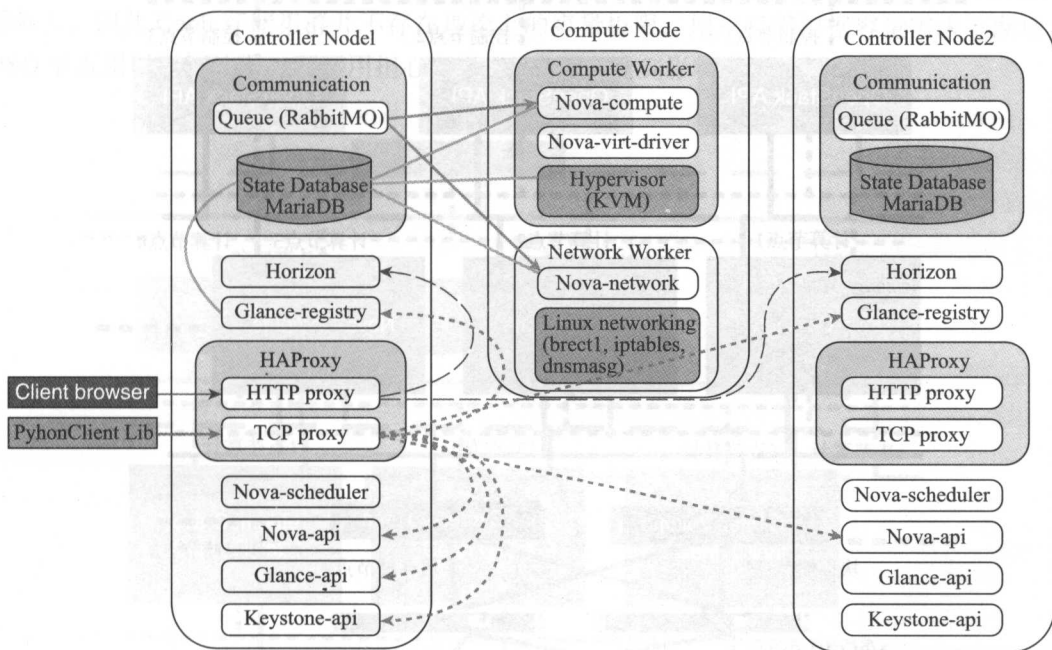


图 11-8 Mirantis OpenStack 高可用部署架构

在企业生产环境中，采用类似图 11-9 所示的 OpenStack 高可用集群有很多优势，首先冗余控制节点为 OpenStack 集群的控制层面提供了高可靠和可用性，同时将集群服务、负载均衡服务、消息队列服务、数据库服务和缓存服务等与计算无关的 OpenStack 基础服务与计算节点独立并集中到高可用控制节点统一部署管理。在以后的使用过程中，用户对集群的扩容仅限于计算和存储，而计算节点的扩容完全是 Scale-out 形式的，且独立于控制节点和后端存储阵列，计算节点扩容也仅需针对 Nova-compute 服务即可。此外，由于采用了传统企业级存储阵列，存储空间的扩容也变得极为简单，只需在线对存储阵列进行磁盘扩容即可，并且无须改变任何 OpenStack 集群配置。用户还可以利用 VPLEX/SVC 将数据同步到多个底层存储阵列，从而为关键数据提供更高层次的保护。采用类似图 11-9 所示的传统企业级存储架构的 OpenStack 高可用集群的不足之处在于成本较高，复杂的后端底层存储和 SAN 网络需要专门的存储管理员维护，并且多数存储阵列的扩容仍然是 Scale-up 形式（也有类似 IBM XIV 的 Scale-out 阵列）。在大容量扩容之后，有限的存储控制器处理能力和前后端带宽必然限制高并发用户对海量存储的访问，因此是否选用此类型的 OpenStack 高

可用集群架构，需要用户根据自己的成本和业务慎重考虑。

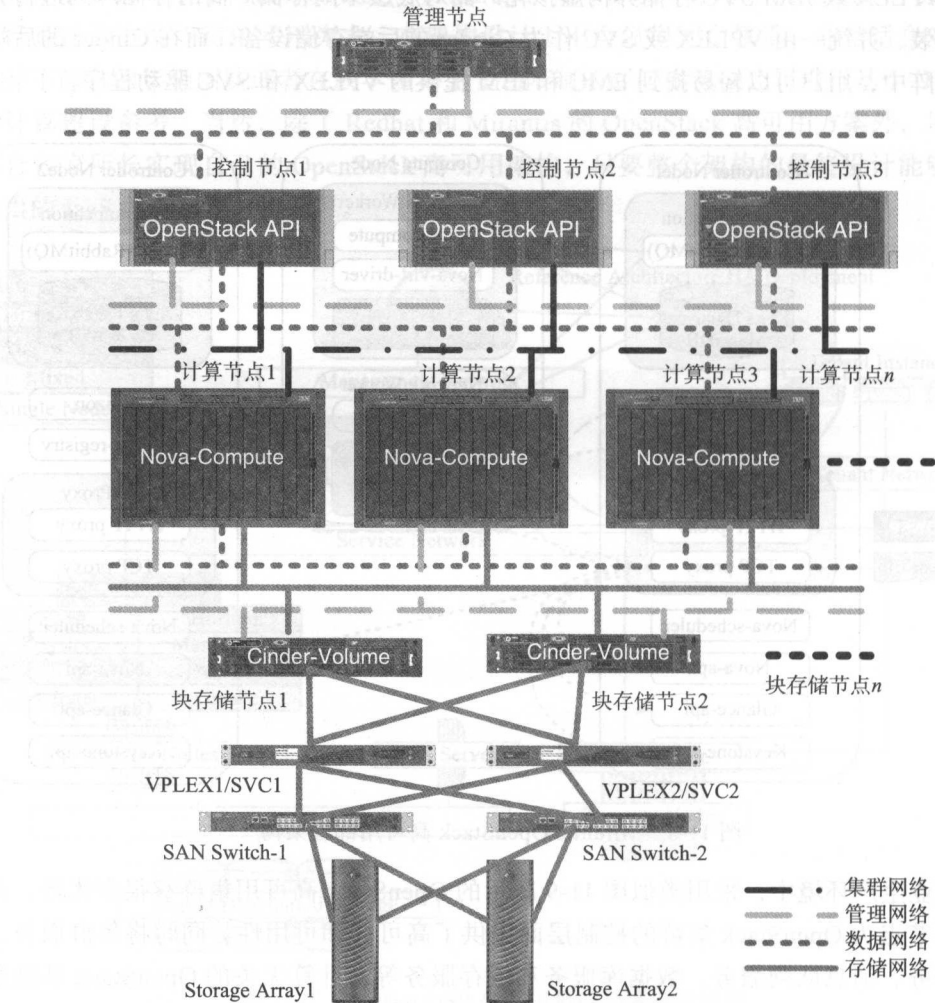


图 11-9 基于 Cinder 块存储的企业级 OpenStack 高可用私有云架构

与图 11-9 不同，图 11-10 采用的是基于分布式开源存储集群 Ceph 的 OpenStack 高可用架构。Ceph 作为一种统一分布式存储集群，其高可用、高可靠和故障自我愈合方面都有相当不错的表现，并且也是 OpenStack 中使用极为普遍的后端存储之一，目前其提供的文件系统存储 CephFS、块存储 RBD 和对象存储 RGW 均为稳定版本。Ceph RBD 不仅可以作为 Cinder 的块存储后端，还可以为 Glance 和 Nova 提供镜像存储和虚拟机临时磁盘空间。Ceph 存储集群主要由 Ceph 监控节点和 Ceph OSD 节点组成，其中监控节点负责整个集群的健康检查并保存集群运行状态和提供集群数据恢复的依据；而 OSD 节点负责用户数据的存储。Ceph 监控节点对整个 Ceph 集群的正常运行至关重要，可采用仲裁机制来保证监控

节点的高可用，因此生产环境下建议监控节点数目最少为 3 个。图 11-10 中所示 Ceph 监控节点被放置到 OpenStack 控制节点上，当然用户可以使用独立的 Ceph 监控节点，或者将其部署到 OSD 节点上（为了实现控制与数据的分离，不建议 OSD 节点同时作为监控节点）。与传统企业级 Scale-up 扩容的存储阵列不同，Ceph OSD 节点可以按需在线以 Scale-out 的形式扩容。从理论上讲，Ceph OSD 节点越多，Ceph 集群的可用性、可靠性和存储性能就越强大，因此 Ceph 存储集群并不存在理论上的容量极限，用户只需根据容量需求不断增加 OSD 节点用以存储数据，而不用担心 Ceph 客户端会受到影响。

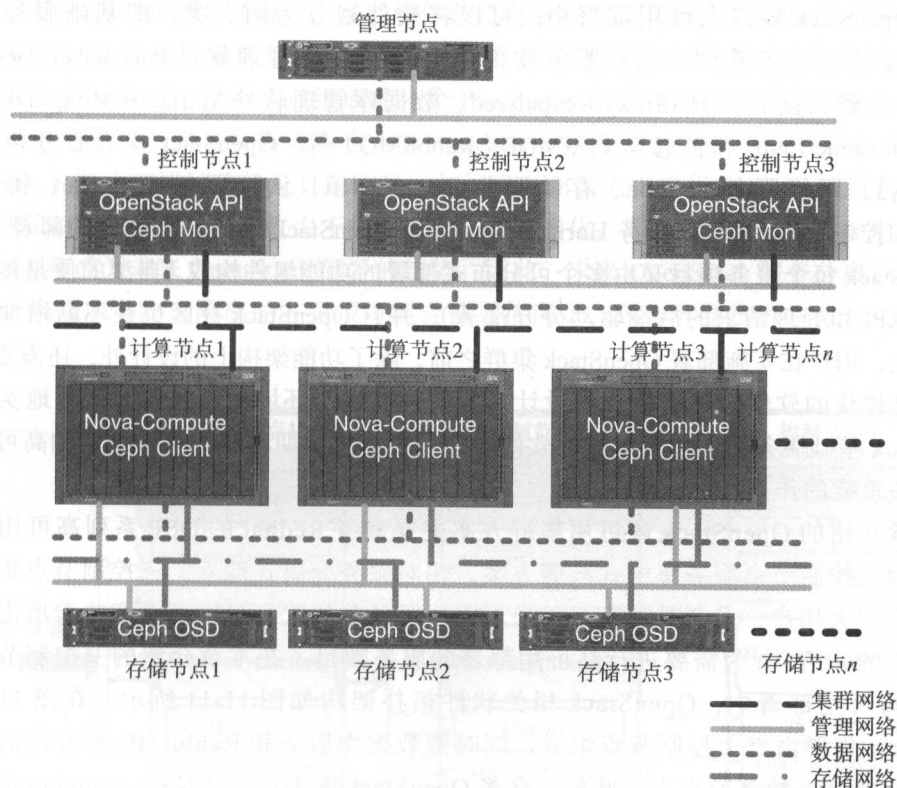


图 11-10 基于 Ceph 存储集群的企业级 OpenStack 高可用私有云架构

在生产环境中，采用基于 Ceph 存储集群的 OpenStack 高可用集群架构具有很多优势：首先，Ceph 是一种开源分布式的存储集群软件，采用 Ceph 作为 OpenStack 集群的后端存储，在成本上相对传统企业级动辄上百万元的存储阵列要实惠很多；其次，Ceph 天生具备大规模集群高并发访问的优势，其节点越多优势越明显，非常适合大规模和海量数据存储的业务场景；Ceph 存储集群完全采用 Scale-out 的形式扩容，避免了用户在存储扩容时性能瓶颈的顾虑；此外，Ceph 提供的文件系统存储、块存储和对象存储可以同时满足用户不同的数据存储需求；其强壮的稳定性和数据自我恢复能力也是 Ceph 存储集群特有优势。不过，相对传统企业级存储阵列和 SAN 存储网络，Ceph 依然很年轻，而且在 Ceph 较长

一段时间的发展过程中,虽然其理论依据已经非常成熟,但是其科研院校出身背景使得其在企业关键业务系统数据存储领域的使用却很少。正是 OpenStack 的出现带来了 Ceph 在企业应用中的普及,因此企业在使用 Ceph 存储集群为 OpenStack 提供后端存储时,需要投入相当多的人力对 Ceph 进行持续的研究跟进,方能解决突发的 Ceph 存储集群故障问题。

11.2.3 OpenStack 高可用部署软件拓扑架构

在 OpenStack 集群高可用部署中,可以将软件划分为两大类,即基础服务软件和 OpenStack 核心服务软件。基础服务软件主要包括集群管理软件 Pacemaker/Pacemaker_remote、负载均衡软件 HAProxy/Keepalived、数据库管理软件 MariaDB/MongoDB、缓存系统 Memcache/Redis 和消息队列 AMQP (RabbitMQ) 等。OpenStack 核心服务包括计算服务 Nova、网络服务 Neutron、存储服务 Cidner/Swift、认证服务 Keystone、镜像服务 Glance 和控制面板 (GUI) 服务 Horizon 等,由于 OpenStack 服务具有极高的部署灵活性,而 OpenStack 每个服务项目又由多个可分布式部署的功能组件构成 (典型的便是接收请求的前端 API 和处理请求的后端驱动分开部署),并且 OpenStack 社区也在不断增加新的项目。因此,用户在实施部署 OpenStack 集群之前,除了功能架构上的设计外,还需要对提供各个功能模块的软件拓扑进行规划设计,尤其是在生产环境中,如何无死角地实现全部 OpenStack 基础服务软件和核心服务软件的高可用,对后期集群的整体稳定和高可用运行起着至关重要的作用。

本章介绍的 OpenStack 高可用集群方案主要参考 Redhat 的 OSP 系列高可用部署架构,采用三控制节点服务集中式部署方案,相对服务分离式部署,三控制节点集中式部署更适合广大用户,尤其是私有云用户。因为服务分离式高可用部署意味着所需服务器数目为 3 的 n 倍, n 为需要进行高可用部署的服务数目。在本章介绍的三控制节点服务集中式高可用部署中,OpenStack 相关软件拓扑架构如图 11-11 所示。在图 11-11 中,Pacemaker 集群由三个控制节点组成,同时像数据库服务和 RabbitMQ 服务的高可用集群均运行在三个控制节点上,此外,众多 OpenStack 服务组件 (Nova-compute 运行在独立的计算节点上) 也运行在三个控制节点上,并由 Pacemaker 资源管理器进行控制管理。Pacemaker 集群中的每个资源均被指派一个虚拟服务 IP,客户端通过虚拟 IP 对集群资源进行访问,同时访问请求被 HAProxy 负载均衡器根据设定的负载均衡算法转发到三个控制节点上。

图 11-11 是从控制节点角度给出的软件部署拓扑架构图,将高可用部署的全部软件拆分之后,OpenStack 高可用集群软件部署模式如图 11-12 所示。其中 C1、C2 和 C3 分别代表图 11-11 中所示的三个控制节点,图 11-12 所示形象地给出了 OpenStack 集群中服务软件高可用部署的分布情况和数据访问流向。Pacemaker 集群管理着若干虚拟服务 IP,每个虚拟 IP 均代表了某种集群服务资源,而集群服务资源均以 Active/Active 或者 Active/

Passive 高可用模式运行在三个控制节点上。同时，任何对集群资源的访问均需要通过运行在三个控制节点上的 HAProxy 负载均衡器，并由负载均衡器将服务请求转发到三个控制节点中的某个节点上进行响应处理。从理论上讲，图 11-12 中所示的每个集群服务均可独立部署到三个节点组成的独立集群中，而 HAProxy 负载均衡软件也可由硬件负载均衡交换机替换，如 F5 公司的负载均衡设备。如此一来，OpenStack 高可用集群就会由若干个 Pacemaker 集群组成，每个 Pacemaker 集群由三个节点构成并且仅运行一个服务，集群服务之间在物理层面被完全隔离，对于大型公有云部署，这是 RedHat 推荐的高可用部署模式，但是对于私有云部署而言，这种软件独立部署模式无疑极大地增加了部署成本，同时也增加了运维管理的集群数目。而实际上，在控制节点物理资源充分的情况下，将全部服务集成部署到三个控制节点组成的一个 Pacemaker 集群中，也可满足 OpenStack 集群高可用服务的要求，而且所需成本也可被大多数用户接受。此外，随着容器技术的发展，OpenStack 容器化部署的趋势也日趋明显，如国内九州云大力推行的 Kolla 项目和 Marantis 推行的 Kubernetes (k8s) 都在致力于将 OpenStack 容器化，即采用 Kolla 或 K8s 来编排部署 OpenStack 将成为未来 OpenStack 部署的趋势。因此，未来 OpenStack 服务分离到独立集群中部署以实现服务物理隔离的方案很可能会被直接淘汰，因为容器技术的特点之一便是服务隔离。

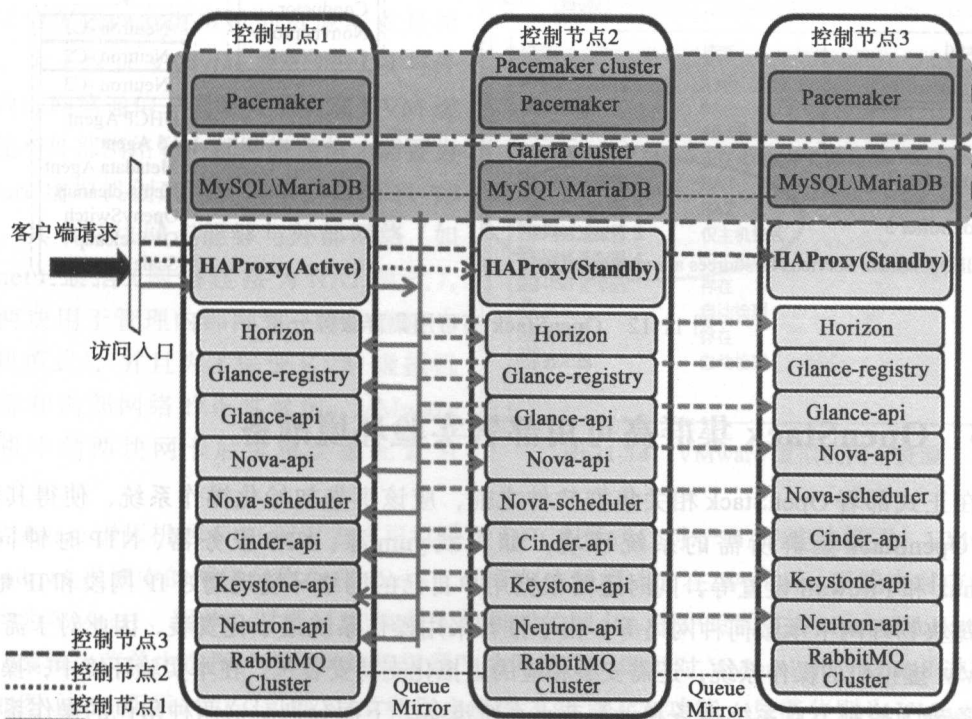


图 11-11 OpenStack 高可用集群软件部署拓扑架构

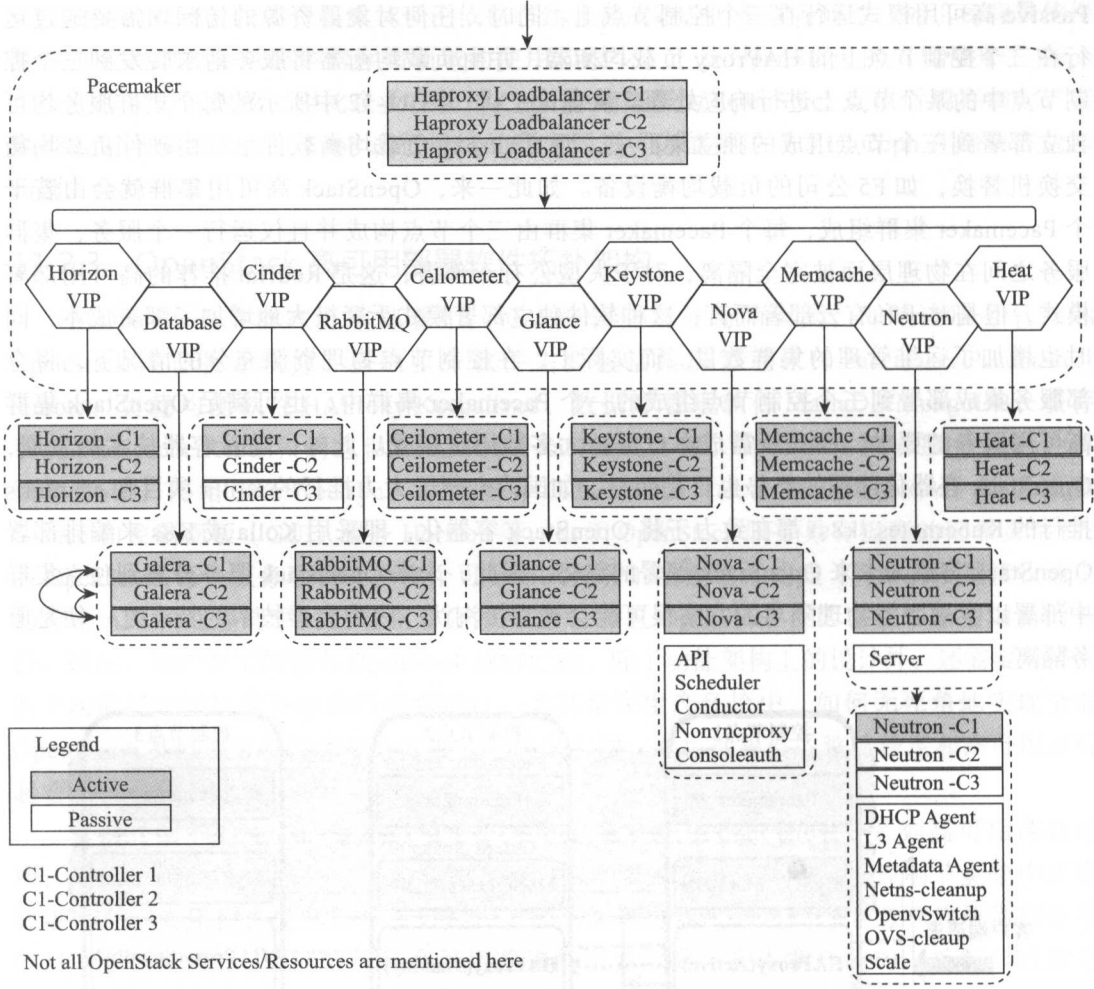


图 11-12 OpenStack 高可用集群服务分布

11.3 OpenStack 集群高可用部署实验环境准备

在正式部署 OpenStack 相关集群软件之前，应该事先初始化操作系统，使得其满足运行 OpenStack 集群所需的系统环境，如离线 yum 源、NFS 服务器、NTP 时钟同步、SELinux 和 Firewall 设置等，同时还需根据用户自己的网络环境规划好 IP 网段和 IP 地址，以及每块物理网卡承载何种网络等。由于推荐采用操作系统最小化安装，因此对于需要创建 KVM 虚拟机的操作系统，还需安装相应的虚拟化支持安装包。在本实验环境中，操作系统准备分为控制节点系统准备和计算节点系统准备。下面分别对这两种角色的操作系统进行初始化准备。

11.3.1 控制节点 VMware 宿主机准备

1. VMware 虚拟机创建

在本实验中，为了实现控制节点的 Fencing 功能，控制节点由三个位于 VMware 虚拟机中的 KVM 虚机构成，VMware 虚拟机安装 Centos7.2 操作系统，作为控制节点的 KVM 虚拟机同样安装 Centos7.2 操作系统。为了实现控制节点高可用，实验环境中 OpenStack 高可用集群控制节点由独立的三台 VMware 虚拟机提供，如图 11-13 所示，每台 VMware 虚拟机分别命名为 controller1、controller2 和 controller3。OpenStack 高可用集群的三台控制节点将分别由图 11-13 中所示的三台 VMware 虚拟机各自创建的 KVM 虚拟机承担，并将控制节点 KVM 虚拟机分别命名为 controller1-vm、controller2-vm 和 controller3-vm。

三台 VMware 虚拟机的资源配置是相同的，为了保证 VMwrae 虚拟机可以与外部和内部网络通信，同时为了实现 KVM 虚拟机能与外部网络和内部网络通信，需要为 VMware 虚拟机创建四块网卡，如图 11-14 所示，其中两块网卡能够与外部网络（如 Internet）通信（网路连接为 NAT 模式），另外两块用于管理内部网络（网络连接为仅主机模式），并且为了保证 KVM 虚拟机与外部和内部网络的正常通信，VMware 虚拟机中的两块网卡后续将会被配置为网桥。

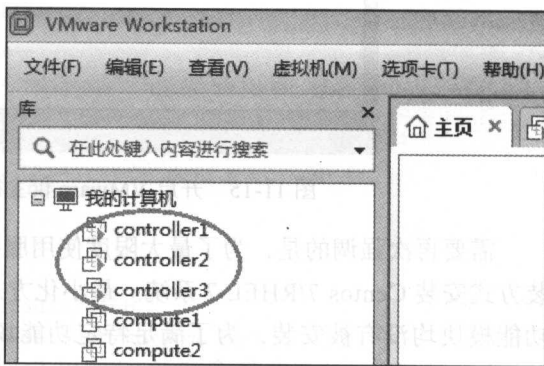


图 11-13 提供控制节点 KVM 虚拟机的三台 VMware 宿主虚拟机

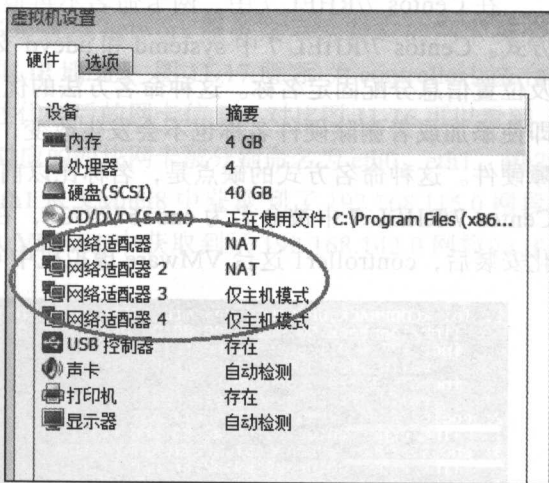


图 11-14 VMware 虚拟机网卡资源

VMware 虚拟机网卡 IP 配置方式可以采用 DHCP 形式。对于 NAT 连接模式，VMware 在 Windows 系统中的网络名称为 VMnet8；对于仅主机模式，在 Windows 系统中对应的网络名称为 VMnet1。因此，对于 NAT 和仅主机模式的网卡，用户只需在 VMware Workstaion 中的“编辑”菜单中选择“虚拟网络编辑器”功能选项，即可对 VMnet1 和 VMnet8 进行编辑，如图 11-15 所示。为了实现 NAT 和仅主机模式的 DHCP 功能，只需分别为 VMnet1 和 VMnet8 开启 DHCP 功能，并设置对应的 DHCP 网段即可。

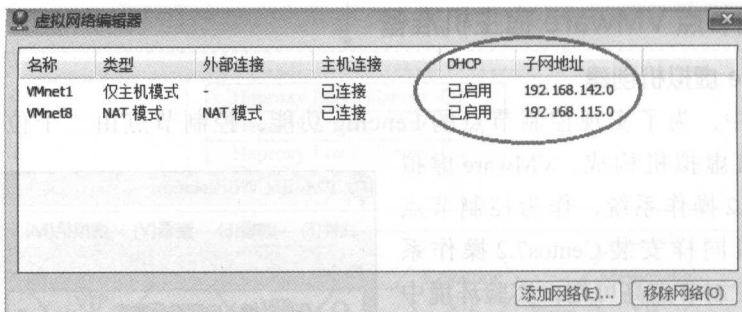


图 11-15 开启 VMware 仅主机和 NAT 模式的 DHCP 功能

需要再次强调的是,为了最大限度使用服务器资源,VMware 虚拟机建议采用最小化安装方式安装 Centos 7/RHEL 7 系统。最小化方式安装速度非常快,但是很多针对特定场景的功能模块均没有被安装,为了满足特定功能场景需求,后续用户需要自己手动安装所需的软件包。

2. 网卡重命名

在 Centos 7/RHEL 7 中,网卡命名不再遵从原来的命名规则,而是启用了最新的命名方式。Centos 7/RHEL 7 中 systemd 和 udevd 支持不同的命名方案,默认是根据固件、拓扑及位置信息分配固定名称。这种命名方法的优点是全自动且可预测后续新增设备的名称,即使添加或者删除硬件名称也不会发生改变,而且可以在不产生任何影响的情况下替换故障硬件。这种命名方式的缺点是,名称比以前使用的更难读写,如以前的网卡名称 eth0 在 Centos 7/RHEL 7 中被命名为 eno16777736,在读写引用时非常不便。图 11-16 所示是最小化安装后,controller1 这台 VMware 虚拟机中 Centos7.2 操作系统中的网卡名称。

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: eno16777736: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 00:0c:29:b9:a2:85 brd ff:ff:ff:ff:ff:ff
3: eno33554960: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 00:0c:29:b9:a2:8f brd ff:ff:ff:ff:ff:ff
4: eno50332184: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 00:0c:29:b9:a2:99 brd ff:ff:ff:ff:ff:ff
5: eno67109408: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 00:0c:29:b9:a2:a3 brd ff:ff:ff:ff:ff:ff
```

图 11-16 VMware 虚拟机 controller1 中的网卡命名情况

为了便于记忆和使用,建议在正式使用系统之前将 Centos 7/RHEL 7 这种网卡命名方式改为之前的 ethx 形式。对于大规模集群节点而言,手工更改显然不现实,读者可以参考如下代码对集群节点进行批量更改^①。运行如下代码之后,节点中全部用户网卡都会被重新命名,即由原来的 enoxxxx 现实变为 ethx 形式。

① 代码参见 https://github.com/ynwssjx/Openstack-HA-Deployment/blob/master/change_default_nic_name.sh。


```

#/usr/bin/bash
#this script will change default ifcfg-enxxxxx to ifcfg-ethx
ls -l /etc/sysconfig/network-scripts|awk '/ifcfg-eno[0-9]*/ {print \$9}' > default_
  nic_name.txt
i=0
cat default_nic_name.txt| while read line
do
    cd /etc/sysconfig/network-scripts
    name=$(echo $line|cut -b 7-)
    cp $line ${line}.bak
    sed -i "s/$name/eth${i}/g" $line
    sed -i 's/ONBOOT=no/ONBOOT=yes/g' $line
    mv $line ifcfg-eth${i}
    i=$(expr $i + 1)
done
sed -i '/GRUB_CMDLINE_LINUX=/d' /etc/default/grub
echo 'GRUB_CMDLINE_LINUX="rd.lvm.lv=centos/root rd.lvm.lv=\
centos/swap crashkernel=auto net.ifnames=0 biosdevname=0 rhgb \
quiet"' >> /etc/default/grub
grub2-mkconfig -o /boot/grub2/grub.cfg
reboot

```

上述代码在系统重启后生效，并且重启系统或 network 服务之后，对应的网卡将会从 VMware 的 VMnet1 和 VMnet8 中自动获取 IP 地址。图 11-17 所示为 controller1 这台 VMware 虚拟机中的 Centos7.2 系统在运行上述代码后的网卡信息。对比图 11-16 可以看到，运行上述代码重命名网卡之后，controller1 中对应的四块网卡被分别命名为 eth0、eth1、eth2 和 eth3，同时对应 NAT 连接模式的 eth0 和 eth1 从 VMnet8 中获取到了 192.168.115.0 网段的 IP 地址，对应仅主机模式的 eth2 和 eth3 从 VMnet1 中获取到了 192.168.142.0 网段的 IP 地址。

```

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 00:0c:29:b9:a2:85 brd ff:ff:ff:ff:ff:ff
    inet 192.168.115.20/24 brd 192.168.115.255 scope global dynamic eth0
        valid_lft 1741sec preferred_lft 1741sec
    inet6 fe80::20c:29ff:feb9:a285/64 scope link
        valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 00:0c:29:b9:a2:8f brd ff:ff:ff:ff:ff:ff
    inet 192.168.115.21/24 brd 192.168.115.255 scope global dynamic eth1
        valid_lft 1741sec preferred_lft 1741sec
    inet6 fe80::20c:29ff:feb9:a28f/64 scope link
        valid_lft forever preferred_lft forever
4: eth2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 00:0c:29:b9:a2:99 brd ff:ff:ff:ff:ff:ff
    inet 192.168.142.21/24 brd 192.168.142.255 scope global dynamic eth2
        valid_lft 1741sec preferred_lft 1741sec
    inet6 fe80::20c:29ff:feb9:a299/64 scope link
        valid_lft forever preferred_lft forever
5: eth3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 00:0c:29:b9:a2:a3 brd ff:ff:ff:ff:ff:ff
    inet 192.168.142.22/24 brd 192.168.142.255 scope global dynamic eth3
        valid_lft 1741sec preferred_lft 1741sec
    inet6 fe80::20c:29ff:feb9:a2a3/64 scope link
        valid_lft forever preferred_lft forever

```

图 11-17 VMware 虚拟机 controller1 中的网卡重命名结果

3. 设置主机名

主机名称设置很简单，对于三台 VMware 虚拟机，主机名称分别设置为 controller1、controller2 和 controller3。主机名需要一个对应的主机 IP 以便进行地址解析，设置主机名之后，将对应的主机名和 IP 写入 /etc/hosts 配置文件中，具体如下：

```
hostnamectl set-hostname ${NAMEHOST}
echo "${host_ip} ${NAMEHOST}" >>/etc/hosts
```

4. 设置防火墙

对于公有云而言，安全设置是非常重要的环节，尤其是防火墙的设置，需要细化到各个服务和端口。对于私有云环境而言，防火墙显得不是很重要，在启用防火墙的情况下，如果对防火墙规则不是很熟悉，反而会影响很多服务的正常启动和运行。因此在本节中，建议直接关闭防火墙，具体如下：

```
systemctl stop firewalld.service
systemctl disable firewalld.service
```

5. 设置 SELinux

SELinux 是一种通过上下文标签来严格设置访问权限的安全机制。SELinux 提供了一种灵活的强访问控制 (MAC) 系统，且内嵌于 Linux Kernel 中。SELinux 定义了系统中每个用户、进程、应用和文件的访问和转变的权限，然后它使用一个安全策略来控制用户、进程、应用和文件这些实体之间的交互，安全策略指定如何严格或宽松地进行检查。SELinux 对系统用户是透明的，系统管理员需要考虑在他的服务器中如何制定严格的策略。策略可以根据需要设置为严格或宽松策略。在启用 SELinux 后，只有同时满足了标准 Linux 访问控制和 SELinux 访问控制时，主体才能访问客体。在 Centos 7/RHEL 7 中，SELinux 配置文件为 /etc/sysconfig/selinux 或 /etc/selinux/config，其默认状态为 enforcing，同时还有 permissive 和 disable 两个状态。通过 getenforce 命令可以获取 SELinux 当前状态；通过 setenforce 0 命令可以将 SELinux 由 enforcing 设置为 permissive；通过 setenforce 1 命令可以将 SELinux 由 permissive 设置为 enforcing。在本节中，建议将 SELinux 设置为 disable 以避免不必要的麻烦。永久性的设置需要更改 SELinux 配置文件并且重启系统，具体如下：

```
[root@controller1 ~]#sed -i "s/^SELINUX=enforcing/SELINUX=disabled/g"\
/etc/selinux/config
[root@controller1 ~]#reboot
```

6. 设置 NTP

对集群而言，很多莫名其妙的问题都可以追溯至节点时间不同步这种低级错误上，因此在正式使用部署集群应用之前，首先需要将集群各个节点的时间进行同步。对于私有云环境而言，集群节点可能没法同步到互联网上的时钟源，因此可以将集群中某个节点的时钟作为集群时钟源来同步，使得集群全部节点时钟相对该时钟源节点的时钟是同步的。本

节中，将 controller1 作为 NTP 时钟源，controller2、controller3、compute1、compute2、controller1-vm、controller2-vm 和 controller3-vm 均与 controller1 保持时钟同步。NTP 客户端节点只需将 controller1 配置为时钟源即可，为了保证 NTP 时钟同步，可以设置一个 crontab 任务，每 10 分钟主动发起一次时钟同步，如下：

```
[root@controller2~]#echo "*/10 * * * * /usr/sbin/ntpdate ${NTP_SERVER}\  
2>&1> /tmp/ntp.log;hwclock -w"
```

在 NTP 时钟同步之后，controller1、controller2 和 controller3 之间应该具有相同的日期和时间，如果同时对三个节点运行 date 命令，则可以看到如下所示相同的日期和时间：

```
[root@controller1 ~]# date  
Mon Nov 28 22:46:36 CST 2016  
[root@controller3 ~]# date  
Mon Nov 28 22:46:36 CST 2016  
[root@controller2 ~]# date  
Mon Nov 28 22:46:36 CST 2016
```

7. 配置本地 yum 源

对于离线安装，yum 源的配置至关重要。在配置本地 yum 源之前，用户需要事先根据 11.1 节中的介绍将 RedHat 发行版本 RDO 的 RPM 包同步到本地。在本节中，controller1 同时充当管理节点和 NFS 服务器，因此在配置 yum 源之前，务必将 Centos7.2 操作系统镜像以及 OpenStack 相关 RPM 包及其依赖包全部上传到 controller1 中指定的目录下。在本节环境中，默认将 Centos7.2 ISO 镜像安装包上传到 /data/ISO 目录中，将 RDO OpenStack 安装包分别上传到 /data/rdo-openstack-kilo/openstack-common、/data/rdo-openstack-kilo/openstack-kilo 和 /data/rdo-openstack-epel 目录中^①，之后运行位于 Github 中的 Master 控制节点初始化脚本（脚本名称为 openstack_master_ctr-nodes_prepare.sh）^②。脚本将会自动创建本地 yum 仓库，并自动配置指向本地 yum 仓库的 yum 源文件。脚本运行完成之后，用户即可离线安装 OpenStack 及其依赖包。

对于其他节点，只需将 controller1 节点通过 NFS 导出的 /data 目录和 /etc/yum.repos.d 目录挂载到本地即可，即其他节点通过 NFS 共享 controller1 节点的本地 yum 源。在 controller1 节点上配置完成本地 yum 源后，将会出现如下的 yum 源配置文件，在后续的软件安装过程中，全部通过如下 yum 本地源进行离线安装，而不会去访问 Internet。

```
[root@controller1 ~]# cd /etc/yum.repos.d  
[root@controller1 yum.repos.d]# ls -l  
total 16  
-rw-r--r--. 1 root root 77 Nov 28 22:02 centos7-iso.repo  
-rw-r--r--. 1 root root 128 Nov 28 22:05 openstack-common.repo  
-rw-r--r--. 1 root root 122 Nov 28 22:05 openstack-kilo.repo
```

① RDO Openstack Kilo 版本的安装包可以到如下地址下载：<http://pan.baidu.com/s/1o8B9rVc>

② 参见 https://github.com/ynwsjx/Openstack-HA-Deployment/blob/master/openstack_master_ctr-nodes_prepare.sh

```
-rw-r--r--. 1 root root 109 Nov 28 22:05 rdo-epel.repo
```

对于其他节点，将通过 NFS 的形式共享 controller1 的 yum 本地源，即集群中全部节点均共享 controller1 的 yum 源，因此集群全部节点均可实现离线安装。

8. 配置 NFS

NFS 是集群管理中不可缺少的配置，尤其是在静态共享资源方面，NFS 有着得天独厚的优势。在本节的 OpenStack 集群部署中，controller1 被配置为 NFS 服务器，并将其 /data 和 /etc/yum.repos.d 目录共享出去，以便其他集群节点可以通过 mount 形式挂载共享 controller1 的本地 yum 源。对于其他节点，为了实现关机重启后自动挂载 /data 和 /etc/yum.repos.d 目录，需要将挂载条目写入 /etc/fstab 文件中。NFS 配置完成后，在 controller1 上将会看到 NFS 导出的目标目录：

```
[root@controller1 ~]# showmount -e
Export list for controller1:
/etc/yum.repos.d *
/data *
```

在任意一个 NFS 客户端，NFS 配置完成后，可以看到 controller1 的 /data 和 /etc/yum.repos.d 目录已经被挂载到本地：

```
root@controller2 ~]# df -h
Filesystem                Size  Used Avail Use% Mounted on
controller1:/etc/yum.repos.d 36G   17G   19G   48% /etc/yum.repos.d
controller1:/data          36G   17G   19G   48% /data
```

9. 配置 libvirt

由于系统采用最小化安装，虚拟化相关功能不被支持，为了后续创建 KVM 虚拟机，需要手动安装 Libvirt 相关软件包并启动 Libvirtd 服务。在启用 Libvirtd 服务后，系统主机中会自动生成 virbr0 虚拟交换机，默认情况下主机中的 KVM 虚拟机均通过该虚拟交换机以 NAT 形式（即 IP 伪装，IP Masquerade）与外界通信。由于本节实验中采用桥接形式（Bridge）直接将 KVM 虚拟机与外界进行通信，因此不会用到 virbr0，故将其关闭。对于 Intel 处理器，支持虚拟化的 Linux 系统内核中需要加载 KVM 和 KVM_intel 模块。如果系统没有自动加载，则需进行手动加载。

10. 配置网桥

本节实验环境中的 KVM 虚拟机将利用 VMware 宿主机中的网桥直接与外界通信。因此在创建 KVM 虚拟机之前，需要在 VMware 宿主机中创建对应的网桥。KVM 虚拟机有一个外部网络和一个内部管理网络，因此在 VMware 宿主机中创建一个外网网桥和一个内网网桥。外网网桥绑定能与外界通信的 eth0 物理网卡，内网网桥绑定能与内网通信的 eth3 物理网卡。网桥名称用户可自定义，当网桥设置成功之后，与网桥绑定的 eth0 和 eth3 物理网卡将不会被配置 IP 地址，IP 地址被配置到网桥上。

外网和内网网桥创建完成后，系统中将会对应新增两个网络接口，通过 `brctl` 命令可以查看到网桥对应的物理网络接口，如下所示：

```
[root@controller1 ~]# brctl show
bridge name      bridge id      STP enabled      interfaces
ext_br0          8000.000c29b9a285  no               eth0
inter_br0        8000.000c29b9a2a3  no               eth3
[root@controller1 ~]# ip a
.....
8: ext_br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 00:0c:29:b9:a2:85 brd ff:ff:ff:ff:ff:ff
    inet 192.168.115.100/24 brd 192.168.115.255 scope global ext_br0
        valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:feb9:a285/64 scope link
        valid_lft forever preferred_lft forever
9: inter_br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 00:0c:29:b9:a2:a3 brd ff:ff:ff:ff:ff:ff
    inet 192.168.142.100/24 brd 192.168.142.255 scope global inter_br0
        valid_lft forever preferred_lft forever
    inet6 fe80::f099:90ff:feld:9a55/64 scope link
        valid_lft forever preferred_lft forever
.....
```

11. 常用工具软件包安装

最小化安装时，很多常用的系统工具均不能使用，如网络配置查看命令行工具 `ifconfig`、网络诊断命令 `tcpdump`、网络文件共享 `NFS` 和 `SMB` 等服务均无法使用。为了便于日常维护和快速操作，可以适当补充安装常用工具包：

```
[root@controller1 ~]#yum install fence* bind-utils tcpdump expect \
sos nfs-utils cifs-utils net-tools -y
```

在实际部署中，上述全部配置过程可以通过脚本自动完成。具体脚本可以参考 Github 上的开源代码，地址为 <https://github.com/ynwssjx/Openstack-HA-Deployment>，对应的配置脚本为 `openstack_master_ctr-nodes_prepare.sh`，用户只需在 `controller1` 节点上运行 `openstack_master_ctr-nodes_prepare.sh` 脚本。该脚本将会自动完成上述介绍的集群节点初始化过程。

11.3.2 控制节点 KVM 虚拟机准备

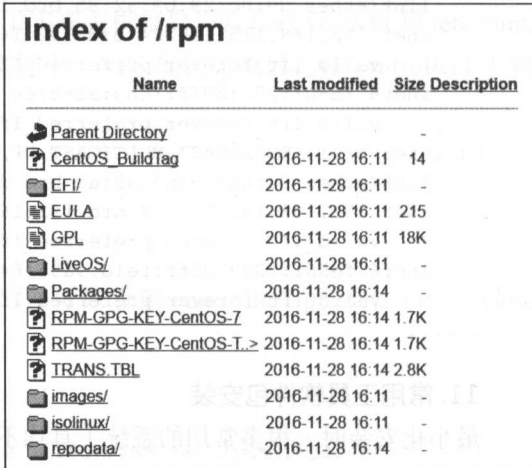
在本节的实验环境中，OpenStack 高可用集群控制节点由三台 KVM 虚拟机组成（`controller1-vm`、`controller2-vm` 和 `controller3-vm`），KVM 虚拟机的宿主机为前一节创建的运行 Centos7.2 的三台 VMware 虚拟机（`controller1`、`controller2` 和 `controller3`）。本节将重点介绍如何在 Centos/RHEL 系统中创建 KVM 虚拟机。本节内容也可作为 OpenStack 系统镜像创建的参考^①。

① 更多 KVM 虚拟化技术请参考任永杰和单海涛所著的《KVM 虚拟化技术：实战与原理解析》。

1. KVM 虚拟机安装源设置

在使用 virt-install 安装 KVM 虚拟机之前，需要配置安装源，即为 virt-install 指定安装包位置仓库（通过 --location 参数指定）。本节实验环境使用 Centos 7.2 系统镜像来安装 KVM 虚拟机。在 11.3.1 节中，Centos 7.2 的 ISO 镜像已经被解压到 controller1 节点的 /data/ISO 目录，并且 controller2 和 controller3 节点也通过 NFS 挂载到本地 /data 目录，即每台 VMware 虚拟机的 /data/ISO 目录都存放了 ISO 安装镜像源。本节以在 controller1 节点创建 KVM 虚拟机为例进行讲解，其他节点创建过程与此类似。为简单起见，此处配置 HTTP 下载服务器供 virt-install 在安装时进行软件包下载，HTTP 服务器的具体配置过程可参考 create_vm_on_master_node.sh 脚本^①中的 configure_apache_server() 函数。该函数运行完成后，通过 http://controller1_ip/rpm/ 即可下载安装包。本例中 controller1 的 IP 为 192.68.142.21，因此在浏览器中输入 http://192.168.142.21/rpm/ 即可下载 HTTP 站点软件包，如图 11-18 所示。

同样，在 virt-install 的安装命令中，为 --location 参数指定 http://192.168.142.21/rpm/，则 virt-install 将自动到 HTTP 站点提取安装包。另外，/data 目录并不具备 httpd_sys_content 上下文标签，因此如果启用 SELinux，则 SELinux 会禁止访问该目录，用户将收到权限不允许的提示。通常情况下建议关闭 SELinux（重启生效），在 SELinux 被禁用的情况下将不会出现此类情况。如果确实要使用 SELinux，则可以使用 semanage 命令为 /data 目录设置允许 httpd 访问的标签。



Name	Last modified	Size	Description
Parent Directory		-	
CentOS_BuildTag	2016-11-28 16:11	14	
EFI/	2016-11-28 16:11	-	
EULA	2016-11-28 16:11	215	
GPL	2016-11-28 16:11	18K	
LiveOS/	2016-11-28 16:11	-	
Packages/	2016-11-28 16:14	-	
RPM-GPG-KEY-CentOS-7	2016-11-28 16:14	1.7K	
RPM-GPG-KEY-CentOS-7.2	2016-11-28 16:14	1.7K	
TRANS.TBL	2016-11-28 16:14	2.8K	
images/	2016-11-28 16:11	-	
isolinux/	2016-11-28 16:11	-	
repodata/	2016-11-28 16:14	-	

图 11-18 下载 HTTP 站点安装包

2. KVM 虚拟机 Fencing 设置

作为试验环境，Fencing 功能可能并不重要，但是如果作为生产环境，则 Fencing 功能是必须的。本节的实验环境中，之所以要在 VMware 虚拟机中创建 KVM 虚拟机来作为 OpenStack 高可用集群的控制节点，就是希望借用现有的 KVM 虚拟机 Fencing 驱动来模拟控制节点的 Fencing 功能。如果是生产环境，则控制节点通常由物理服务器承载。常见的物理服务器均具备 Fencing 功能，因此生产环境中创建 KVM 虚拟机来作为控制节点这个步骤是完全没有必要的。在本实验环境中，通过 fence-virt 软件对 KVM 虚拟机进行 Fencing 操作，fence-virt 的配置可参考 create_vm_on_master_node.sh 脚本中的 configure_fence_file() 函数实现。该函数执行完成后，在 VMware 宿主虚拟机上即可通过 fence_xvm 命令

① https://github.com/ynwssjx/Openstack-HA-Deployment/blob/master/create_vm_on_master_node.sh

来验证配置是否正确。fence_xvm 命令可以列出 (list) 主机上全部虚拟机信息, 同时获取全部虚拟机的状态, 并对指定虚拟机进行关机 (off)、启动 (on)、重启 (reboot) 和状态获取 (status) 等操作。fence_xvm -o list 命令将列出宿主机上全部虚拟机及其状态, 验证结果如下:

```
[root@controller1 ~]# fence_xvm -o list
controller1-vm          b9baabdf-9822-447e-95c0-35ff1af0d5ed  on
[root@controller1 ~]# virsh list --all
 Id      Name                               State
-----
 4       controller1-vm                    running
```

此外, 可以通过 fence_xvm 命令测试对虚拟机的 Fencing 操作, 如下命令将会重启 VMware 宿主机 controller3 上的 KVM 虚拟机 controller3-vm:

```
[root@controller3 ~]# fence_xvm -o reboot -H controller3-vm
```

通过 VNC 客户端可以观察到, 在运行 fence_xvm -o reboot 命令后, KVM 虚拟机 controller3-vm 系统自动重启, 如图 11-19 所示。

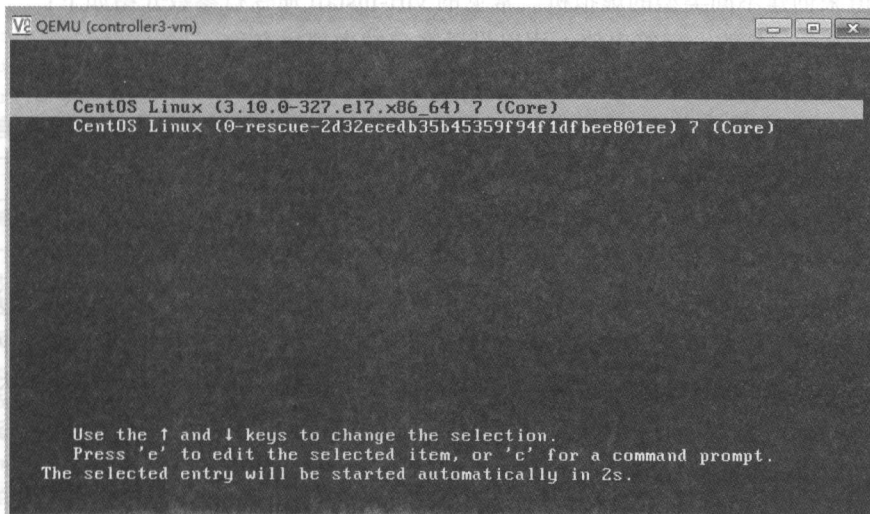


图 11-19 fence_xvm 重启 KVM 虚拟机 controller3-vm

3. KVM 虚拟机自定义安装

在制作适用于特定环境的系统镜像时, 并不希望安装社区或厂商发行的全部软件包, 因为社区或厂商发行的系统镜像 (如 CentOS-7-x86_64-DVD-1511.iso) 包含了很多不必要的软件包, 这些不必要的软件包使得制作出来的镜像显得十分臃肿, 非常不利于云环境中镜像的快速分发。因此在制作镜像时通常采取自定义安装方式来选择性地安装所需软件和工具包。在本实验环境中, 我们事先制作了一个 KVM 虚拟机自定义安装初始化文件 virt-

base.ks。该文件将告知 virt-install 需要安装的软件工具包，同时告知其在系统安装完成后需要对 KVM 虚拟机做哪些系统初始化配置工作，如设置 /etc/hosts、/etc/fstab 等系统配置文件，以及设置主机名、本地 yum 源仓库和配置网卡 IP 地址等。virt-base.ks 自定义安装文件的制作可参考 create_vm_on_master_node.sh 脚本中的 create_guest_inject_script() 函数来实现，该函数执行完成后，将会在当前目录生成一个名为 virt-base.ks 的自定义安装文件，virt-base.ks 文件通过 --initrd-inject 参数和 --extra-args 参数传入 virt-install 命令，virt-install 命令将根据 virt-base.ks 文件指定的内容和命令进行 KVM 虚拟机操作系统的自定义安装和初始化设置。在采用 virt-base.ks 进行自定义 KVM 虚拟机安装时，virt-install 除了安装 Kernel 模块外，还会自动安装 nfs-utils、net-tools、ntp 和 wget 等实用工具包，同时根据脚本对系统进行初始化设置，并挂载 NFS 文件系统、配置 yum 源以及设置网卡 IP 地址等。

4. 创建 KVM 虚拟机镜像

在 Linux 系统中，创建 KVM 虚拟机最常使用的便是 virt-install 命令行工具。virt-install 命令行工具具有复杂多样的参数供用户选择，用户可以通过 virt-install 提供的参数设置不同的值来创建不同类型的虚拟机。常见的 virt-install 命令行参数介绍如下：

- ❑ -n|--name 参数：字符串值，用以设置客户端虚拟机名称。
- ❑ -r|--ram 参数：整数值，用以设置为客户端虚拟机分配的内存。
- ❑ -u|--uuid 参数：字符串值，用以设置客户机 UUID，用户未指定时系统会自动生成。
- ❑ --vcpu 参数：整数值，用以设置客户机的 vcpu 个数。
- ❑ -v|--hvm 参数：没有值，用以指定客户机虚拟化模式为全虚拟化。
- ❑ -p|--paravirt 参数：没有值，用以指定客户机虚拟化模式为半虚拟化。
- ❑ -l|--location 参数：字符串值，用以指定安装源的位置，有本地安装源和远程安装源之分。远程安装源包括 nfs、http 和 ftp 几种方式，常用于 kickstart 网络安装中，--location 参数允许使用 --extra-args 和 --initrd-inject 来指定额外的内核参数。
- ❑ -c|--cdrom 参数：字符串值，用于指定光驱安装介质路径，一般为 iso 格式镜像文件，如 --cdrom=/root/centos7.iso。
- ❑ --disk 参数：字符串值，用以指定存放客户机操作系统的介质路径，类似 Windows 系统安装中使用的 C 盘，如 --disk=/image/centos7/centos7-base.img。
- ❑ -w|--network 参数：值为网络名称，用以指定客户机连接到的主机网络，对于想要使用静态网络的客户机，通常将该参数值设置为主机网桥（BRIDGE）。
- ❑ --cpuset 参数：整数列表，设置哪些物理 CPU 能够被虚拟客户机使用。
- ❑ --os-type 参数：其值为操作系统类型（如 Linux、Windows），用以指定客户机操作系统类型，虚拟化引擎将根据不同类型的操作进行客户机系统优化。
- ❑ --os-variant 参数：其值为特定的操作系统（如 Fedora18、Rhel6、Rhel7、Winxp 和 Win2k3 等），虚拟化引擎将针对该参数设置的操作系统进一步优化虚拟机配置。

- ❑ **--accelerate** 参数：用以指定 KVM 或 KQEMU 内核加速，这个选项目前是默认。
- ❑ **--initrd-inject** 参数：字符串值，此参数通常与 **--location** 以及 **--extra-argsc** 参数配合使用，主要用以 kickstart 自动安装场景中，用以指定 ks 文件的绝对路径。
- ❑ **--extra-args** 参数：字符串值，在通过 **--location** 参数安装客户机时，用以指定附加内核命令行参数给安装程序。在 kickstart 自动安装中，通常与 **--initrd-inject** 参数配合使用，如 **--initrd-inject=/path/to/my.ks --extra-args "ks=file:/my.ks"**。
- ❑ **--graphics** 参数：字符串值，用以指定使用的图形安装界面，此参数并不会配置客户机的虚拟硬件，而是设置可以通过什么方式访问客户机。如 **--graphics vn,listen='0.0.0.0'**，这样便可通过 VNC 客户端来访问 KVM 虚拟机。

在本实验环境中，先通过 **qemu-img** 创建一个 **qcow2** 格式的镜像文件，然后通过 **virt-install** 命令工具安装 KVM 虚拟机操作系统，使用远程 HTTP 安装源，通过 **--initrd-inject** 和 **--extra-args** 参数将 ks 文件传递给安装程序。创建的虚拟机名称为 **centos7-base**（对应的虚拟机镜像文件为 **centos7-base.img**）。在 **virt-install** 安装过程中，可以使用 VNC 客户端查看安装进度。**virt-install** 命令行工具安装 KVM 虚拟机的过程可参考如下脚本^①：

```
#/usr/bin/bash
yum install -y virt-install
mkdir -p $vm_dir
//创建qcow2格式的磁盘文件
qemu-img create -f qcow2 -o preallocation=metadata \
${vm_dir}/${vm_base} $vm_disk
//创建KVM虚拟机
virt-install --connect=qemu:///system \
--network=bridge:$int_br_name,mac=$int_mac \
--initrd-inject=./virt-base.ks \
--extra-args="ks=file:/virt-base.ks console=tty0 \
console=ttyS0,115200 serial rd_NO_PLYMOUTH" \
--name=centos7-base \
--disk path=${vm_dir}/${vm_base},format=qcow2,cache=none \
--ram $vm_ram \
--vcpus=$vm_cpu \
--check-cpu \
--accelerate \
--os-type linux \
--os-variant rhel7 \
--hvm \
--location=$rpm_http \
--graphics vnc,listen='0.0.0.0' \
--noautoconsole
```

① 也可参考 **create_vm_on_master_node.sh** 脚本中的 **create_base_vm()** 函数。

上述 KVM 虚拟机安装脚本中, 通过 `--graphics` 参数设置虚拟机访问方式为虚拟网络客户端 (VNC, Virtual Network Console), 并监听全部 IP。这里没有指定 VNC 的监听端口, 而 VNC 默认监听端口为 5900, 因此只需在 VNC 客户端软件中指定主机 IP 和 5900 端口, 即可访问主机上的 KVM 虚拟机。VNC 客户端访问方式如图 11-20 所示。

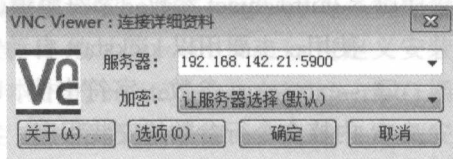


图 11-20 VNC Viewer 访问 KVM 虚拟机

图 11-20 中使用的 VNC Viewer 是一款常用的免费 VNC 客户端软件, 通过 VNC 访问 KVM 虚拟机后, 即可看到 KVM 虚拟机安装启动过程。安装完成之后, 通过 VNC 客户端即可实现 KVM 虚拟机的终端命令行操作, VNC 客户端对 centos7-base 虚拟机的访问结果如图 11-21 所示。

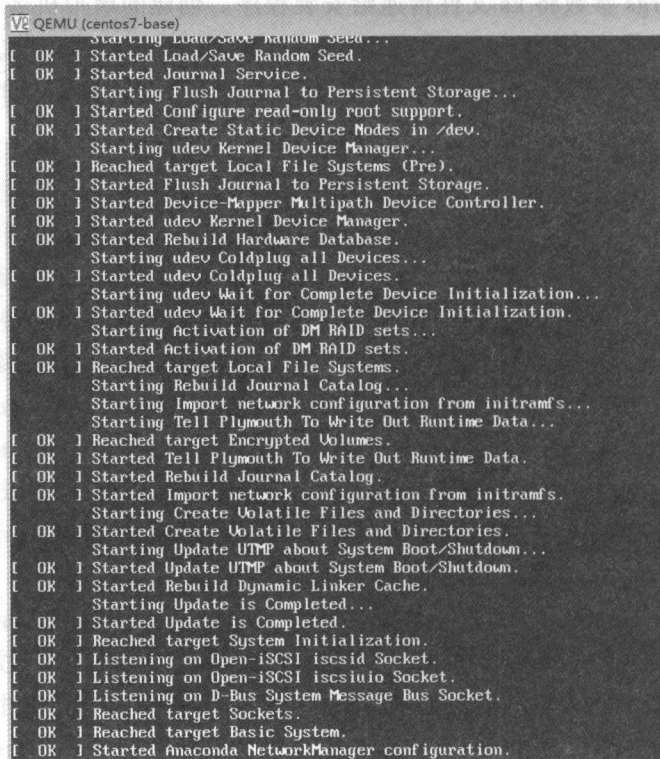


图 11-21 VNC 客户端访问 KVM 虚拟机创建过程

5. 定义 KVM 虚拟机配置文件

虚拟机的运行有多种方式, 用户既可以使用 `qemu-system-x86` 或者 `qemu-kvm` 来运行虚拟机, 也可以使用 Libvirt 的 `virsh` 命令从 XML 文件来定义和运行虚拟机。Libvirt 使用 XML 文件对虚拟机进行配置, 其中包括虚拟机名称、分配内存、`vcpu` 和网络等多种信

息,通常 XML 文件又称虚拟机配置文件。使用 `qemu-kvm` 来运行虚拟机,与 `virt-install` 命令创建 KVM 虚拟机一样,需要在命令行指定很多虚拟机运行参数,包括内存、`vcpu`、网络和镜像文件等参数,而这些参数实际上都可以通过 XML 语言定义到 XML 文件中,并通过 Libvirt 提供的 `virsh` 命令操作 XML 文件从而操作虚拟机。在 Libvirt 中,定义虚拟机通常也称为定义一个 Domain,在实际应用中,通常不必去修改 `virt-install` 生成的虚拟机镜像系统文件,而是将该镜像系统文件作为一个基础镜像,并在其上创建一个仅保存差异部分的差分镜像文件,同时通过 XML 语言重新定义虚拟机配置,并以 `virsh` 命令来定义和启动虚拟机。用这种方式对虚拟机进行操作非常灵活,并且不会更改原始虚拟机镜像(除非使用 `commit` 命令)。如果想要重新定义虚拟机资源配置,只需修改 XML 文件并重新定义和启动虚拟机即可。在本节的实验环境中,基础镜像文件为 `virt-install` 创建的 `centos7-base.img` 文件,这里将其拷贝到 `/localvms` 目录,并在该目录中创建一个虚拟机配置临时文件 `template.xml`,之后以 `centos7-base.img` 和 `template.xml` 为基础,创建一个差分镜像文件 `${vm_name}.img` 和一个定义虚拟机的 XML 文件 `${vm_name}.xml`,并通过 `virsh` 命令来定义和启动虚拟机。另外,此处启动的虚拟机即是最终作为 OpenStack 高可用集群控制节点的 KVM 虚拟机。虚拟机的具体定义和启动过程可参考 `create_vm_on_master_node.sh` 脚本中的 `define_vm_xml()` 函数。

该函数执行完成后,在主机 `/localvms` 目录下将生成如下文件,同时通过 `virsh` 命令可以看到虚拟机的运行状态:

```
[root@controller1 localvms]# ls -l
total 1925344
-rw-r--r-- 1 qemu qemu 21478375424 Nov 28 23:40 centos7-base.img
-rw-r--r-- 1 qemu qemu 86507520 Nov 29 15:07 controller1-vm.img
-rw-r--r-- 1 root root 1192 Nov 28 23:40 controller1-vm.xml
-rw-r--r-- 1 root root 1192 Nov 28 23:40 template.xml
[root@controller1 localvms]# virsh list --all

```

Id	Name	State
4	controller1-vm	running

`virsh` 命令启动虚拟机后,即可通过 VNC 客户端访问虚拟机。进入虚拟机系统命令行后,可以检查虚拟机的系统配置是否与自定义安装文件设置的一致,如图 11-22 所示。

本节创建 KVM 虚拟机的源代码参见 <https://github.com/ynwssjx/Openstack-HA-Deployment>,其中,实现 KVM 虚拟机创建和初始化的脚本主要包括 `create_vm_on_all_nodes.sh`、`create_vm_on_master_nodes.sh` 和 `create_vm_on_slave_nodes.sh` 三个脚本,但是在实际执行中,只需运行第一个脚本 `create_vm_on_all_node.sh`,该脚本会自动在三台运行 Centos 系统的 VMware 虚拟机上分别创建三个 KVM 虚拟机。需要指出的是,使用物理服务器作为控制节点的生产环境并不需要执行本节所做的操作,本节内容仅针对采用虚拟机作为 OpenStack 高可用集群控制节点的实验环境。

```

[root@controller1-vm ~]# df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/mapper/centos-root 10G  913M  8.1G   9% /
devtmpfs        1.4G     0  1.4G   0% /dev
tmpfs           1.4G     0  1.4G   0% /dev/shm
tmpfs           1.4G  8.3M  1.4G   1% /run
tmpfs           1.4G     0  1.4G   0% /sys/fs/cgroup
/dev/xda1       497M  125M  372M  26% /boot
controller1:/data 36G  20G  16G  56% /data
tmpfs           278M     0  278M   0% /run/user/0

[root@controller1-vm ~]# yum repolist
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
repo id                               repo name                               stat
centos7-iso                           centos7-iso                            3.72
openstack-common                       openstack common packages              1.01
openstack-kilo                         openstack kilo packages                28
rdo-epel                               extra packages enterprise linux       8.33
repolist: 13,351

[root@controller1-vm ~]# more /proc/cpuinfo | grep -i processor
processor       : 0
processor       : 1
processor       : 2

[root@controller1-vm ~]# more /proc/meminfo | grep -i mem
MemTotal:      2042464 kB
MemFree:       2548576 kB
MemAvailable:  2624792 kB
Shmem:         8488 kB

```

图 11-22 通过 VNC 访问 controller1-vm 虚拟机并查看系统配置情况

11.3.3 计算节点 VMware 虚拟机准备

在本节实验环境中，计算节点由运行 Centos7.2 Linux 系统的 VMware 虚拟机直接承担，实验环境仅设置两台计算节点。计算节点主要运行 Pacemaker_remote 集群软件和 Nova-compute 组件。计算节点接受 OpenStack 高可用集群控制节点的控制管理，在任意计算节点故障情况下，位于其上的虚拟机应该实现自动撤离，并在另一正常计算节点上重新启动，即 OpenStack 虚拟机应该具备高可用性。因为与控制节点 VMware 虚拟宿主主机初始化类似，因此相关的计算节点初始化步骤可以参考 11.3.1 节的相关内容。计算节点系统初始化准备工作也包括网卡重命名、主机名设置、NFS 配置、NTP 配置以及本地 yum 源设置等。由于计算节点采用最小化安装方式，为了支持 Nova 创建 KVM 虚拟机，还需补充安装 Libvirt 相关软件包。此外，计算节点需要加入 Pacemaker 集群，因此还需在计算节点安装 Pacemaker_remote 等相应的集群软件。

在计算节点系统初始化完成后，整个 OpenStack 集群节点系统初始化工作已经完成，因此在计算节点初始化完成的同时可以对全部集群节点需要共享的文件进行本地同步，如主机解析文件 /etc/hosts 及 Pacemaker 集群授权访问密钥等。对于集群系统而言，最好保证全部节点具有相同的主机解析文件以便节点之间可以任意解析通信，同时 Pacemaker 集群节点之间需要保证相同的 Pacemaker 密钥（位于 /etc/pacemaker 目录），以便集群可以正常运行。对于存在大量计算节点的 OpenStack 集群，手动初始化计算节点显然是极为不现实的，因此当规模较小时，可以考虑采取基于 SSH 的脚本自动化部署方式；而在生产环境规模较大时，可能需要考虑更为专业的集群部署工具，如 Puppet、Chef、Ansible 等。

本节介绍的计算节点初始化脚本可以参考笔者位于 Github 网站上的 OpenStack 高可用集群开源部署代码 (<https://github.com/ynwssjx/Openstack-HA-Deployment>), 其中用于计算节点初始化的脚本是 `openstack_computer_node_initial_main.sh` 脚本, 该脚本会对 OpenStack 集群中的全部计算节点进行初始化设置。

11.4 OpenStack 高可用集群基础服务部署

11.4.1 Pacemaker 集群管理软件部署

Pacemaker 是 OpenStack 高可用集群的大脑, 负责对全部基础软件服务和 OpenStack 核心组件服务的资源调度和管理, 因此在 OpenStack 高可用集群部署中, 第一步要完成的工作便是在控制节点上部署 Pacemaker 集群。更多关于 Pacemaker 集群资源管理器的工作原理和使用方法请参考第 3 章。本节重点介绍如何在三节点的控制节点集群中安装部署 Pacemaker, 而关于本节介绍的 Pacemaker 自动部署代码, 可以参考笔者位于 github 的开源项目 `Openstack-HA-Deployment`^①。在本章介绍的 OpenStack 高可用集群部署环境中, 控制节点为运行在 VMware 虚拟机上的 KVM 虚拟机, 而 KVM 虚拟机的系统在镜像创建时已通过脚本注入的形式进行了部分初始化设置, 因此本节在 KVM 虚拟机控制节点中部署 Pacemaker 时, 仅需做部分系统设置, 在正式创建 Pacemaker 集群之前, 需要在控制节点系统中进行以下的准备工作。

安装集群相关软件包:

```
yum install -y pcs pacemaker corosync fence-agents-all resource-agents
```

启动 pcsd 进程:

```
systemctl enable pcsd.service
systemctl start pcsd.service
```

禁用 Firewalld 和 SELinux

```
systemctl disable firewalld.service
systemctl stop firewalld.service
setenforce 0
```

配置时钟同步服务器 NTP:

```
sed -i s/^server.*/ /etc/ntp.conf
echo "server $ntp_server iburst" >> /etc/ntp.conf
echo "SYNC_HWCLOCK=yes" >> /etc/sysconfig/ntpdate
systemctl enable ntpd.service
systemctl start ntpd.service
```

① 具体地址为 <https://github.com/ynwssjx/Openstack-HA-Deployment>

设置 Fencing 设备密钥 key:

```
mkdir -p /etc/cluster
echo ${fence_xvm} > /etc/cluster/fence_xvm.key
```

由于控制节点 controller1-vm、controller2-vm 和 controller3-vm 之间需要进行无密钥 SSH 信任互访, 因此需要建立节点之间的 SSH 信任关系。对于脚本自动化部署, 可以通过 expect 工具和 ssh-copy-id 命令实现。假设 nodes_list 变量代表了全部控制节点主机名列表, 则节点互信脚本可参考如下实现:

```
for i in `echo $nodes_list|awk '{print $2,$3}'`
do
expect << EOF
    set timeout 2
    spawn ssh-copy-id root@$i
    expect {
        "yes/no" {send "yes\r";exp_continue}
        "password" {send "root\r";exp_continue}
    }
EOF
done
```

在 shell 脚本中, expect 工具通常用于应对交互式的命令行输出问题, 而 ssh-copy-id 会自动将本地节点的公钥分发到目标节点上。Pacemaker 集群软件安装完成后, 默认在系统中自动创建一个名为 hacluster 的集群用户。为了将节点加入集群, 需要为各个控制节点的 hacluster 用户设置密码:

```
echo ${hacluster_passwd}|passwd --stdin hacluster
```

这里的 \${hacluster_passwd} 为用户指定的密码, 设置完成之后, 便可在任意控制节点上创建 Pacemaker 集群。在创建过程中, 需要为集群节点进行授权, 同时需要指定集群名称和需要加入集群的节点主机名:

```
pcs cluster auth $nodes_list -u hacluster -p ${hacluster_passwd} --force
pcs cluster setup --force --name $ha_cluster_name $nodes_list
pcs cluster enable --all
pcs cluster start --all
```

为了实现对控制节点的 Fencing 操作, 需要为 Pacemaker 集群配置 Stonith 资源。根据 11.3.2 节介绍的 KVM 虚拟机 Fencing 配置, Pacemaker 的 Stonith 资源创建命令如下:

```
pcs stonith create fence1 fence_xvm multicast_address=225.0.0.1
pcs stonith create fence2 fence_xvm multicast_address=225.0.0.2
pcs stonith create fence3 fence_xvm multicast_address=225.0.0.3
pcs resource defaults resource-stickiness=INFINITY
```

Pacemaker 集群和 Stonith 资源创建完成后, 在任何一个控制节点均可以查看 Pacemaker 集群及其资源运行状态。pcs status 命令将会显示集群及其资源的详细信息。对

于本节创建的 Pacemaker 集群，将会看到如下集群状态信息：

```
[root@controller1-vm ~]# pcs status
Cluster name: openstack-ha
Last updated: Tue Nov 29 22:05:21 2016      Last change: Tue Nov 29 22:05:01
2016 by root via cibadmin on controller1-vm
Stack: corosync
Current DC: controller1-vm (version 1.1.13-10.el7-44eb2dd) - partition with quorum
3 nodes and 3 resources configured
Online: [ controller1-vm controller2-vm controller3-vm ]
Full list of resources:
```

```
fence1 (stonith:fence_xvm):   Started controller1-vm
fence2 (stonith:fence_xvm):   Started controller2-vm
fence3 (stonith:fence_xvm):   Started controller3-vm
```

PCSD Status:

```
controller1-vm: Online
controller2-vm: Online
controller3-vm: Online
```

Daemon Status:

```
corosync: active/enabled
pacemaker: active/enabled
pcsd: active/enabled
```

从控制节点 controller1-vm 的集群状态信息中可以看出，集群当前 DC 为 controller1-vm，集群共有三个成员节点（controller1-vm、controller2-vm 和 controller3-vm）和三个资源（stonith 资源），集群名称为 openstack-ha，三个节点全部为 online 状态。在 OpenStack 高可用集群，尤其是生产环境中，节点隔离（Fencing）对于集群数据的完整性是非常重要的，因此建议在部署后续 OpenStack 相关服务之前，事先对 Pacemaker 集群的 Stonith 资源进行有效性验证。节点隔离有效性可以通过 pcs stonith 命令进行验证，具体过程可参考如下方式进行。

显示当前 Pacemaker 集群中的 stonith 资源：

```
[root@controller1-vm ~]# pcs stonith show
fence1 (stonith:fence_xvm):   Started controller1-vm
fence2 (stonith:fence_xvm):   Started controller2-vm
fence3 (stonith:fence_xvm):   Started controller3-vm
```

假设需要将 controller3-vm 节点隔离（使用 --off 参数表示关闭节点而不是默认的 reboot），则可以通过如下命令实现：

```
[root@controller1-vm ~]# pcs stonith fence controller3-vm --off
Node: controller3-vm fenced
```

检查 KVM 虚拟机 controller3-vm 是否已被关闭：

```
[root@controller3 ~]# virsh list --all
```

Id	Name	State
-	controller3-vm	shut off

确认 controller3-vm 已经被关闭之后, 通过 pcs stonith confirm 命令告知 Pacemaker 集群资源管理器 controller3-vm 节点已经被隔离:

```
[root@controller1-vm ~]# pcs stonith confirm controller3-vm
Node: controller3-vm confirmed fenced
```

此时再检查 Pacemaker 集群及其资源运行状态, 查看 controller3-vm 是否已经为 offline 状态。正常情况下被隔离后的 controller3-vm 在集群中应该为 offline, 具体如下:

```
[root@controller1-vm ~]# pcs status
Cluster name: openstack-ha
Last updated: Tue Nov 29 22:54:45 2016          Last change: Tue Nov 29 22:05:01
          2016 by root via cibadmin on controller1-vm
Stack: corosync
Current DC: controller2-vm (version 1.1.13-10.el7-44eb2dd) - partition with quorum
3 nodes and 3 resources configured
Online: [ controller1-vm controller2-vm ]
OFFLINE: [ controller3-vm ]
.....
```

至此, OpenStack 高可用集群配置的第一步, 即 Pacemaker 集群创建已经成功完成, 后续只需将与各个 OpenStack 高可用集群相关的服务以 Pacemaker 资源形式添加到集群中即可, 具体部署过程参见后续章节。此外, 本节 Pacemaker 集群自动部署的源代码可参见笔者位于 Github 上的开源项目^①, 对应的部署脚本为 l_create_openstack_pacemaker_cluster.sh。

11.4.2 HAProxy 负载均衡器高可用部署

负载均衡是 OpenStack 高可用集群中非常关键的部分, 可以认为负载均衡器是整个 OpenStack 功能集群的入口, 任何外部客户端或内部组件对集群服务的请求访问都需要通过负载均衡器, 从而实现单点故障时集群功能访问仍可继续进行。关于集群负载均衡系统更为详细的工作原理和使用方法请参考第 4 章的相关内容, 本节将重点介绍如何在 Pacemaker 集群中配置 HAProxy 负载均衡器, 以及如何规划 OpenStack 各个相关功能组件的服务 IP 和其在 HAProxy 配置文件中的设置。

通常, 在 HAProxy 高可用配置中, 虚拟 IP 可以有两种配置方式: 其一为使用单一虚拟 IP 作为集群对外服务 IP, 即负载均衡器所有后端服务共享同一个对外服务 IP; 其二为每个功能服务模块使用自己的虚拟 IP, 即虚拟服务 IP 与负载均衡器后端功能模块一一对应。采用第一种方式的优点是节省服务 IP, HAProxy 配置也相对简单, 缺点在于集群所有

① <https://github.com/ynwssjx/Openstack-HA-Deployment>

服务通过单一虚拟 IP 对外暴露，而单一 IP 仅能配置在单一控制节点上（负载均衡器部署在控制节点上），对于大规模集群高并发访问而言，容易造成单一控制节点负载过高和网络瓶颈问题，此类配置多见于 OpenStack 官方高可用配置网站和 Mirantis 部分高可用方案。如果采用第二种配置方式，则可以将不同的服务虚拟 IP 分布到不同的控制节点上，而访问负载也随之分布到不同节点上，因此对于大规模生产环境而言，第二种负载均衡 IP 配置方式更为合理，RedHat 提出的 OpenStack 高可用方案采用的便是此类负载均衡配置方式。为了更为合理地负载客户端请求，本节介绍的 HAProxy 配置方案采用的是第二类虚拟服务 IP 配置方案。在本章介绍的 OpenStack 高可用配置中，对外服务 IP 网段为管理网段，即 192.168.142.0/24。OpenStack 高可用集群各个服务组件对外服务虚拟 IP 规划如表 11-1 所示。

表 11-1 OpenStack 高可用集群虚拟 IP 规划

服务名称	虚拟 IP	端口
MariaDB	192.168.142.201	3306
RabbitMQ	192.168.142.202	5672
Dashboard	192.168.142.211	80
Keystone-admin	192.168.142.203	35357
Keystone-public	192.168.142.203	5000
Glance-api	192.168.142.205	9191
Glance-registry	192.168.142.205	9292
Cinder	192.168.142.206	8776
Swift	192.168.142.208	8080
Neutron	192.168.142.209	9696
Nova-api	192.168.142.210	8774
Nova-metadata	192.168.142.210	8775
Nova-vnc-xvpxncproxy	192.168.142.210	6081
Nova-vnc-novncproxy	192.168.142.210	6080
heat-cfn	192.168.142.212	8000
heat-cloudw	192.168.142.212	8004
heat-srv	192.168.142.212	8004
ceilometer	192.168.142.214	8777

在配置 HAProxy 之前，需要安装 HAProxy 负载均衡软件包。由于本节介绍的方案将负载均衡器部署在控制节点上，因此还需对控制节点操作系统做相应的设置：

```
yum install -y haproxy
echo "net.ipv4.ip_nonlocal_bind=1" > /etc/sysctl.d/haproxy.conf
sysctl -p
```



```

echo 1 > /proc/sys/net/ipv4/ip_nonlocal_bind
cat >/etc/sysctl.d/tcp_keepalive.conf << EOF
net.ipv4.tcp_keepalive_intvl = 1
net.ipv4.tcp_keepalive_probes = 5
net.ipv4.tcp_keepalive_time = 5
EOF
sysctl net.ipv4.tcp_keepalive_intvl=1
sysctl net.ipv4.tcp_keepalive_probes=5
sysctl net.ipv4.tcp_keepalive_time=5

```

不要手动启动 HAProxy，因为 HAProxy 后续将配置为 Pacemaker 资源，其启动停止等操作完全由 Pacemaker 自行决定。根据表 11-1 中规划设置的服务虚拟 IP，HAProxy 配置文件 `/etc/haproxy/haproxy.cfg` 的配置内容如下所示^①：

```

[root@controller1-vm ~]# more /etc/haproxy/haproxy.cfg
.....
frontend vip-keystone-admin
    bind 192.168.142.203:35357
    default_backend keystone-admin-vms
backend keystone-admin-vms
    balance roundrobin
    server controller1-vm 192.168.142.110:35357 check inter 1s
    server controller2-vm 192.168.142.111:35357 check inter 1s
    server controller3-vm 192.168.142.112:35357 check inter 1s
frontend vip-keystone-public
    bind 192.168.142.203:5000
    default_backend keystone-public-vms
backend keystone-public-vms
    balance roundrobin
    server controller1-vm 192.168.142.110:5000 check inter 1s
    server controller2-vm 192.168.142.111:5000 check inter 1s
    server controller3-vm 192.168.142.112:5000 check inter 1s
frontend vip-glance-api
    bind 192.168.142.205:9191
    default_backend glance-api-vms
backend glance-api-vms
    balance roundrobin
    server controller1-vm 192.168.142.110:9191 check inter 1s
    server controller2-vm 192.168.142.111:9191 check inter 1s
    server controller3-vm 192.168.142.112:9191 check inter 1s
.....

```

在进一步配置之前，务必保证三个控制节点中 `/etc/haproxy/haproxy.cfg` 内容完全一致，关于 `haproxy.cfg` 文件的配置格式和参数解释请参考第 4 章中的相关内容。HAProxy 大致工作原理就是客户端访问前端虚拟 IP，之后 HAProxy 通过一定的负载均衡算法将访问请求负载均衡并转发到后端物理服务器中。上述 `haproxy.cfg` 配置文件中，一共有三个

① 完整配置请参考网址 (<https://github.com/ynwssjx/Openstack-HA-Deployment/blob/master/haproxy.cfg>)

后端物理服务器，即 controller1-vm、controller2-vm 和 controller3-vm，其对应的 IP 分别为 192.168.142.110、192.168.142.111 和 192.168.142.112。在实际应用中，虚拟 IP 可以通过 Keepalived 或者 Pacemaker 资源管理的形式实现高可用，这里采用的虚拟 IP 高可用方案为 Pacemaker 资源形式，即将虚拟 IP 配置为受 Pacemaker 管理的 Ipaddr2 资源，通过 Pacemaker IP 资源管理的形式来实现虚拟 IP 的高可用。在 Pacemaker 集群中，HAProxy 和虚拟 IP 高可用资源的创建过程如下。

创建 Pacemaker 集群 HAProxy 资源：

```
pcs resource create lb-haproxy systemd:haproxy --clone
```

创建 Pacemaker 集群虚拟 IP 资源：

```
components=lb db rabbitmq keystone memcache glance cinder swift-brick\
swift neutron nova horizon heat mongodb ceilometer qpidd
offset=200
internal_network=192.168.142
for section in ${components}; do
    case $section in
        lb|memcache|swift-brick|mongodb)
            echo "No VIP needed for $section"
            ;;
        *)
            pcs resource create vip-${section} IPAddr2\
ip=${internal_network}.${offset} nic=eth1
            pcs constraint order start vip-${section} then lb-haproxy-clone\
kind=Optional
            pcs constraint colocation add vip-${section} with lb-haproxy-clone
            ;;
    esac
    offset=$(( $offset + 1 ))
done
```

虚拟 IP 资源创建完成后，通过 pcs status 命令查看 Pacemaker 集群及其资源运行状态，集群输出信息应该如下：

```
[root@controller1-vm ~]# pcs status
.....
fence3 (stonith:fence_xvm): Started controller3-vm
Clone Set: lb-haproxy-clone [lb-haproxy]
Started: [ controller1-vm controller2-vm controller3-vm ]
vip-db (ocf::heartbeat:IPAddr2): Started controller1-vm
vip-rabbitmq (ocf::heartbeat:IPAddr2): Started controller2-vm
vip-keystone (ocf::heartbeat:IPAddr2): Started controller3-vm
vip-glance (ocf::heartbeat:IPAddr2): Started controller1-vm
vip-cinder (ocf::heartbeat:IPAddr2): Started controller2-vm
vip-swift (ocf::heartbeat:IPAddr2): Started controller3-vm
vip-neutron (ocf::heartbeat:IPAddr2): Started controller1-vm
vip-nova (ocf::heartbeat:IPAddr2): Started controller2-vm
vip-horizon (ocf::heartbeat:IPAddr2): Started controller3-vm
```

```

vip-heat      (ocf::heartbeat:IPaddr2):      Started controller1-vm
vip-ceilometer (ocf::heartbeat:IPaddr2):      Started controller2-vm
vip-qpid      (ocf::heartbeat:IPaddr2):      Started controller3-vm
.....

```

可以看到, HAProxy 以 Clone 资源的形式同时运行在三个控制节点上, 而全部集群虚拟 IP 均匀分布在三个控制节点上。此时, 查看 controller1-vm 中 eth1 网口, 正常情况下应该可以看到 eth1 上除了 192.168.142.110 外, 还被配置了很多虚拟 IP:

```

[root@controller1-vm ~]# ip a
.....
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 00:0c:29:b9:a2:f2 brd ff:ff:ff:ff:ff:ff
    inet 192.168.142.110/24 brd 192.168.142.255 scope global eth1
        valid_lft forever preferred_lft forever
    inet 192.168.142.205/24 brd 192.168.142.255 scope global secondary eth1
        valid_lft forever preferred_lft forever
    inet 192.168.142.201/24 brd 192.168.142.255 scope global secondary eth1
        valid_lft forever preferred_lft forever
    inet 192.168.142.209/24 brd 192.168.142.255 scope global secondary eth1
        valid_lft forever preferred_lft forever
    inet 192.168.142.212/24 brd 192.168.142.255 scope global secondary eth1
        valid_lft forever preferred_lft forever
.....

```

现关闭 controller1-vm 控制节点以模拟该节点故障, 正常情况下之前运行在 controller1-vm 上的虚拟 IP 应该自动迁移到 controller2 或者 controller3 节点上。对 controller1-vm 进行隔离操作:

```

[root@controller3-vm ~]# pcs stonith fence controller1-vm --off
Node: controller1-vm fenced

```

再次检查 Pacemaker 集群上虚拟 IP 资源的节点分布情况。pcs status 输出的集群资源信息如下:

```

[root@controller3-vm ~]# pcs resource
Clone Set: lb-haproxy-clone [lb-haproxy]
    Started: [ controller2-vm controller3-vm ]
    Stopped: [ controller1-vm ]
vip-db (ocf::heartbeat:IPaddr2):      Started controller3-vm
vip-rabbitmq (ocf::heartbeat:IPaddr2): Started controller2-vm
vip-keystone (ocf::heartbeat:IPaddr2): Started controller3-vm
vip-glance   (ocf::heartbeat:IPaddr2): Started controller2-vm
vip-cinder   (ocf::heartbeat:IPaddr2): Started controller2-vm
vip-swift     (ocf::heartbeat:IPaddr2): Started controller3-vm
vip-neutron   (ocf::heartbeat:IPaddr2): Started controller3-vm
vip-nova      (ocf::heartbeat:IPaddr2): Started controller2-vm
vip-horizon   (ocf::heartbeat:IPaddr2): Started controller3-vm
vip-heat      (ocf::heartbeat:IPaddr2): Started controller2-vm
vip-ceilometer (ocf::heartbeat:IPaddr2): Started controller2-vm

```

```
vip-qpid (ocf::heartbeat:IPAddr2): Started controller3-vm
```

可以看到, controller1-vm 节点故障后, Pacemaker 集群 IP 资源一个没少, 即之前位于 controller1-vm 节点的虚拟 IP 资源全部迁移到 controller2-vm 或 controller3-vm 控制节点上, 真正实现了虚拟 IP 单点故障时的高可用。而 HAProxy 由于同时运行在三个控制节点上, 因此任何一个节点故障均不会影响 HAProxy 的正常使用。本节 HAProxy 高可用部署源代码请参考笔者位于 Github 上的开源项目 <https://github.com/ynwssjx/Openstack-HA-Deployment>, 对应的部署脚本为 2_create_vip_resource_on_pacemaker.sh。

11.4.3 MariaDB 关系数据库高可用部署

数据库是 OpenStack 高可用集群最为核心的功能组件之一, 在 OpenStack 云平台中, 关系型数据库 MySQL/MariaDB 存储了用户创建的对象及其状态信息。截至目前, OpenStack 社区使用最多的关系型数据库是 MariaDB, 关于 MariaDB 更多的工作原理与配置使用方式等相关信息, 请参考第 7 章关于集群数据库系统的介绍, 本节主要介绍关系型数据库 MariaDB 在 Pacemaker 集群中的高可用部署。MariaDB 高可用集群部署采用最多也是较为成熟的方案是 MariaDB Galera Cluster, 因此本节将介绍如何在 Pacemaker 集群中配置 MariaDB Galera 集群以实现 OpenStack 集群数据库服务的高可用性。MariaDB 高可用部署过程可参考以下步骤。

安装 MariaDB Galera 集群软件包:

```
yum install -y mariadb-galera-server xinetd rsync
```

mariadb-galera-server 软件包自带有 Galera 集群检查命令 /usr/bin/clustercheck, 要允许 HAProxy 对 Galera 进行健康检查, 但需要先将 Pacemaker 集群中的 HAProxy 资源暂停, 并在 MariaDB 中进行允许健康检查的相关配置:

```
pcs resource disable lb-haproxy
cat > /etc/sysconfig/clustercheck << EOF
MYSQL_USERNAME="clustercheck"
MYSQL_PASSWORD="$password"
MYSQL_HOST="localhost"
MYSQL_PORT="3306"
EOF
systemctl start mysqld.service
//clustercheck命令要正常工作, 需在数据库中创建clustercheck用户
mysql -e "CREATE USER 'clustercheck'@'localhost' IDENTIFIED BY\
'${password}';"
systemctl stop mysqld.service
```

创建 MariaDB Galera 集群之前, 为每个 Galera 集群节点配置集群文件 /etc/my.cnf.d/galera.cnf, 该文件设置了 Galera 集群名称和默认使用的存储引擎等参数, 具体配置可参考如下脚本:

```

cat > /etc/my.cnf.d/galera.cnf << EOF
[mysqld]
skip-name-resolve=1
binlog_format=ROW
default-storage-engine=innodb
innodb_autoinc_lock_mode=2
innodb_locks_unsafe_for_binlog=1
query_cache_size=0
query_cache_type=0
bind_address=$bind_ip
wsrep_provider=/usr/lib64/galera/libgalera_smm.so
wsrep_cluster_name="galera_cluster"
wsrep_slave_threads=1
wsrep_certify_nonPK=1
wsrep_max_ws_rows=131072
wsrep_max_ws_size=1073741824
wsrep_debug=0
wsrep_convert_LOCK_to_trx=0
wsrep_retry_autocommit=1
wsrep_auto_increment_control=1
wsrep_drupal_282555_workaround=0
wsrep_causal_reads=0
wsrep_notify_cmd=
wsrep_sst_method=rsync
EOF

```

在 HAProxy 配置文件中，设置了 `httpchk` 参数以对数据库服务进行周期性的健康检查，数据库服务在负载均衡器 HAProxy 的配置文件 `/etc/haproxy/haproxy.cfg` 中的配置段如下：

```

.....
backend db-vms-galera
    option httpchk
    option tcpka
    stick-table type ip size 1000
    stick on dst
    timeout server 90m
    server controller1-vm 192.168.142.110:3306 check inter 1s port 9200\
    backup on-marked-down shutdown-sessions
    server controller2-vm 192.168.142.111:3306 check inter 1s port 9200\
    backup on-marked-down shutdown-sessions
    server controller3-vm 192.168.142.112:3306 check inter 1s port 9200\
    backup on-marked-down shutdown-sessions
.....

```

为了对数据库服务进行基于 HTTP 协议的健康检查，需要对 `xinetd` 服务进行相关配置，配置文件可在 `/etc/xinetd.d` 目录中自定义创建，注意配置文件中的端口号与 `haproxy.cfg` 中指定的要一致，具体配置参考如下：

```

cat > /etc/xinetd.d/galera-monitor << EOF
service galera-monitor

```



```

{
    port            = 9200
    disable         = no
    socket_type     = stream
    protocol        = tcp
    wait            = no
    user            = root
    group           = root
    groups          = yes
    server          = /usr/bin/clustercheck
    type            = UNLISTED
    per_source      = UNLIMITED
    log_on_success  =
    log_on_failure  = HOST
    flags           = REUSE
}
EOF

```

启动 xinetd 服务，并设置其为开机自启动：

```

systemctl enable xinetd.service
systemctl start xinetd.service

```

现在，可在 Pacemaker 集群中创建 Galera 集群资源。Galera 在 Pacemaker 中以多状态资源 (multi-state) 的形式运行，具体创建命令如下：

```

node_list="$controller1-vm,controller2-vm,$controller3-vm"
pcs resource create galera-galera enable_creation=true\
wsrep_cluster_address="gcomm://{node_list}" \
additional_parameters='--open-files-limit=16384' meta master-max=3\
ordered=true op promote timeout=300s on-fail=block --master

```

Galera 集群资源创建完成后，重新启动 Pacemaker 中的 HAProxy 资源，并设置 Galera 与 HAProxy 之间的顺序约束。在 OpenStack 高可用集群中，所有服务监听的虚拟 IP 均由 HAProxy 负责转发到具体的后端物理服务器上，因此在启动任意后端服务之前，必须先保证 HAProxy 已经成功启动，Galera 集群资源在 Pacemaker 中的启动约束如下：

```

pcs resource enable lb-haproxy
pcs constraint order start lb-haproxy-clone then start galera-master

```

Galera 集群资源创建完成后，可通过 clustercheck 命令检查集群状态是否已经稳定，以及当前节点是否已经同步到 Galera 集群中：

```

[root@controller2-vm ~]# clustercheck
HTTP/1.1 200 OK
Content-Type: text/plain
Connection: close
Content-Length: 32

Galera cluster node is synced.

```

通过 Pacemaker 的集群状态查看命令，也可以看到当前 Pacemaker 集群资源的运行状态，pcs status 命令输出的集群资源信息应该如下：

```
[root@controller2-vm ~]# pcs resource
Clone Set: lb-haproxy-clone [lb-haproxy]
    Started: [ controller1-vm controller2-vm controller3-vm ]
vip-db (ocf::heartbeat:IPaddr2):          Started controller1-vm
vip-rabbitmq (ocf::heartbeat:IPaddr2):      Started controller2-vm
vip-keystone (ocf::heartbeat:IPaddr2):      Started controller3-vm
vip-glance (ocf::heartbeat:IPaddr2):        Started controller1-vm
vip-cinder (ocf::heartbeat:IPaddr2):        Started controller2-vm
vip-swift (ocf::heartbeat:IPaddr2):         Started controller3-vm
vip-neutron (ocf::heartbeat:IPaddr2):       Started controller1-vm
vip-nova (ocf::heartbeat:IPaddr2):          Started controller2-vm
vip-horizon (ocf::heartbeat:IPaddr2):       Started controller3-vm
vip-heat (ocf::heartbeat:IPaddr2):          Started controller1-vm
vip-ceilometer (ocf::heartbeat:IPaddr2):    Started controller2-vm
vip-qpid (ocf::heartbeat:IPaddr2):          Started controller3-vm
Master/Slave Set: galera-master [galera]
    Masters: [ controller1-vm controller2-vm controller3-vm ]
```

Pacemaker 集群中看到的 Galera 集群为 Active/Active 多主集群高可用模式，但是客户端对 Galera 集群的访问是通过 HAProxy 实现的，而在 HAProxy 的后端数据库服务器设置中，客户端对数据库服务器集群的访问仅限制在某个活跃节点，并非同时读写全部节点。为了进一步确认 Galera 集群已经准备就绪，可以进入数据库并查看 wsrep 的相关参数来确认，具体如下：

```
[root@controller1-vm log]# mysql -u root -p
MariaDB [(none)]> show status like 'wsrep%';
```

Variable_name	Value
wsrep_local_state_uuid	c47d5ea9-b668-11e6-b3eb-ebe89b9658d2
wsrep_protocol_version	15
wsrep_last_committed	14
wsrep_replicated	0
wsrep_replicated_bytes	0
wsrep_repl_keys	0
wsrep_repl_keys_bytes	0
wsrep_repl_data_bytes	0
wsrep_repl_other_bytes	0
wsrep_received	12
wsrep_received_bytes	315
wsrep_local_commits	0
wsrep_local_cert_failures	0
wsrep_local_replays	0
wsrep_local_send_queue	0
wsrep_local_send_queue_avg	0.500000
wsrep_local_recv_queue	0

```

| wsrep_local_recv_queue_avg | 0.000000 |
| wsrep_local_cached_downto | 18446744073709551615 |
| wsrep_flow_control_paused_ns | 0 |
| wsrep_flow_control_paused | 0.000000 |
| wsrep_flow_control_sent | 0 |
| wsrep_flow_control_recv | 0 |
| wsrep_cert_deps_distance | 0.000000 |
| wsrep_apply_oooe | 0.000000 |
| wsrep_apply_ool | 0.000000 |
| wsrep_apply_window | 0.000000 |
| wsrep_commit_oooe | 0.000000 |
| wsrep_commit_ool | 0.000000 |
| wsrep_commit_window | 0.000000 |
| wsrep_local_state | 4 |
| wsrep_local_state_comment | Synced |
| wsrep_cert_index_size | 0 |
| wsrep_causal_reads | 0 |
| wsrep_cert_interval | 0.000000 |
| wsrep_incoming_addresses | *.*.110:3306,*.111:3306,*.112:3306 |
| wsrep_cluster_conf_id | 2 |
| wsrep_cluster_size | 3 |
| wsrep_cluster_state_uuid | c47d5ea9-b668-11e6-b3eb-ebe89b9658d2 |
| wsrep_cluster_status | Primary |
| wsrep_connected | ON |
| wsrep_local_bf_aborts | 0 |
| wsrep_local_index | 1 |
| wsrep_provider_name | Galera |
| wsrep_provider_vendor | Codership Oy <info@codership.com> |
| wsrep_provider_version | 3.5(rXXXX) |
| wsrep_ready | ON |
| wsrep_thread_count | 2 |
+-----+-----+
48 rows in set (0.00 sec)

```

通过集群 `wsrep` 参数，可以看到 `wsrep_ready` 已经为 `ON`，表明 Galera 集群已经准备就绪，即现在可对集群数据库进行读写访问。如下代码段将在数据库中创建 OpenStack 各个服务组件，包括与 Keystone、Glance、Cinder、Neutron、Nova 和 Heat 对应的数据库：

```

.....
galera_script=galera.setup
echo "" > $galera_script
echo "GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' IDENTIFIED by 'root' WITH\
GRANT OPTION;" >> $galera_script
for db in keystone glance cinder neutron nova heat; do
    cat<<EOF >> $galera_script
CREATE DATABASE ${db};
GRANT ALL ON ${db}.* TO '${db}'@'%' IDENTIFIED BY '${db}';
EOF
done
echo "FLUSH PRIVILEGES;" >> $galera_script

```

```
mysql mysql < $galera_script
mysqladmin flush-hosts
.....
```

数据库创建命令将会在全部分 Galera 节点上执行，因此创建完成后，可在任意 Galera 节点查看所创建的数据库，具体如下：

```
[root@controller2-vm ~]# mysql -uroot -e "show databases;"
```

```
+-----+
| Database |
+-----+
| information_schema |
| cinder |
| glance |
| heat |
| keystone |
| mysql |
| neutron |
| nova |
| performance_schema |
+-----+
```

```
[root@controller3-vm ~]# mysql -uroot -e "show databases;"
```

```
+-----+
| Database |
+-----+
| information_schema |
| cinder |
| glance |
| heat |
| keystone |
| mysql |
| neutron |
| nova |
| performance_schema |
+-----+
```

```
[root@controller1-vm ~]# mysql -uroot -proot -e "show databases;"
```

```
+-----+
| Database |
+-----+
| information_schema |
| cinder |
| glance |
| heat |
| keystone |
| mysql |
| neutron |
| nova |
| performance_schema |
+-----+
```

可以看到，尽管仅在某个 Galera 节点上执行数据库命令，但是可以在任意节点上看到

数据库命令执行后的结果。至此，关系型数据库 MariaDB 的高可用部署已经完成，用户已经可以对数据库进行高可用读写访问。后续将继续介绍各个 OpenStack 高可用基础服务组件的高可用部署。关于本节介绍的 MariaDB 高可用集群部署源代码，可参考笔者位于 Github 上的开源项目（<https://github.com/ynwssjx/Openstack-HA-Deployment>），对应的部署脚本为 `3_create_mariadb_galera_on_pacemaker.sh`。

11.4.4 Memcache 缓存系统高可用部署

Memcache 缓存主要用于缓解后端数据库查询压力并提高客户端响应速率，Memcache 缓存系统高可用主要通过客户端配置文件来设置，缓存服务器无须特别的集群配置，仅需运行 Memcache 服务即可。关于更多 Memcache 缓存系统的配置使用和工作原理等内容请参考第 6 章集群缓存系统部分，本节仅介绍 Memcache 在 Pacemaker 集群中的资源创建过程。在本节介绍的 OpenStack 高可用部署方案中，Memcache 同时运行在三个控制节点上，即控制节点同时作为 Memcache 缓存系统集群。Memcache 在 Pacemaker 集群中的配置部署很简单，只需在各个控制节点安装 Memcached 软件包，同时将 Memcache 配置为 Pacemaker 的 Clone 资源即可，具体配置过程如下。

在全部控制节点上安装 Memcached 软件包，在集群管理节点上参考如下脚本在全部控制节点上循环安装 Memcached：

```
for (( i = 1; i <=$ha_cluster_node_num; i++ ))
do
    echo "begining install memcached on ${hacluster_node[i]}"
    ssh root@${hacluster_node[i]} "yum install -y memcached"
done
```

其中，`${ha_cluster_node_num}` 为控制节点数目，`${hacluster_node[i]}` 为第 i 个控制节点主机名，上述脚本将在全部控制节点上自动安装 Memcached。Memcached 软件包安装完成之后，在 Pacemaker 集群任意节点上创建 Memcache 资源，具体命令如下：

```
pcs resource create memcached systemd:memcached --clone interleave=true"
```

Memcache 资源创建完成后，各个控制节点上 Memcached 服务的启停将由 Pacemaker 集群资源管理器自动管理，因此不需要手动设置 Memcached 服务的启停。通过 Pacemaker 集群资源查看命令 `pcs resource` 可以看到当前集群中的资源运行情况，具体如下：

```
[root@controller1-vm ~]# pcs resource
Clone Set: lb-haproxy-clone [lb-haproxy]
Started: [ controller1-vm controller2-vm controller3-vm ]
vip-db (ocf::heartbeat:IPaddr2): Started controller1-vm
vip-rabbitmq (ocf::heartbeat:IPaddr2): Started controller2-vm
vip-keystone (ocf::heartbeat:IPaddr2): Started controller3-vm
vip-glance (ocf::heartbeat:IPaddr2): Started controller1-vm
vip-cinder (ocf::heartbeat:IPaddr2): Started controller2-vm
```



```

vip-swift      (ocf::heartbeat:IPAddr2):      Started controller3-vm
vip-neutron    (ocf::heartbeat:IPAddr2):      Started controller1-vm
vip-nova       (ocf::heartbeat:IPAddr2):      Started controller2-vm
vip-horizon    (ocf::heartbeat:IPAddr2):      Started controller3-vm
vip-heat       (ocf::heartbeat:IPAddr2):      Started controller1-vm
vip-ceilometer (ocf::heartbeat:IPAddr2):      Started controller2-vm
vip-qpid       (ocf::heartbeat:IPAddr2):      Started controller3-vm
Master/Slave Set: galera-master [galera]
Masters: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: memcached-clone [memcached]
Started: [ controller1-vm controller2-vm controller3-vm ]

```

可以看到, Memcached 服务同时运行在 controller1-vm、controller2-vm 和 controller3-vm 三个控制节点上。为了实现集群 Memcache 缓存系统的高可用, 用户在配置缓存系统使用客户端时, 只需同时指定 Memcache 缓存服务器为 controller1-vm、controller2-vm 和 controller3-vm 即可, 这样任何一个 Memcached 服务器故障, 集群都可以继续为客户端提供缓存功能。本节介绍的 Memcache 缓存系统高可用部署源代码可参考笔者位于 Github 上的开源项目 (<https://github.com/ynwssjx/Openstack-HA-Deployment>), 对应的部署脚本为 4_create_memcached_resource_on_pacemaker.sh。

11.4.5 RabbitMQ 消息队列高可用部署

消息队列是 OpenStack 高可用集群各个服务组件之间彼此通信的基础, 消息队列中存储了组件之间进行远程过程调用 (RPC) 的数据信息, 因此消息队列系统的高可用是 OpenStack 集群能够提供高可用服务的核心基础。在 OpenStack 社区, 消息队列系统可以由 RabbitMQ 或 ZeroMQ 等开源软件实现, 而 RabbitMQ 是使用最多的高级消息队列系统, 关于 RabbitMQ 更详细的工作原理和使用配置方法请参考第 5 章, 本节主要介绍 RabbitMQ 在 Pacemaker 集群中的高可用配置。在 Pacemaker 中创建 RabbitMQ 资源时, 资源代理 RA 的选择可以有两种方案: 其一为使用 ClusterLabs^①网站提供的 OCF 脚本 rabbitmq-cluster, RedHat 的 OpenStack 高可用方案所采用的 RabbitMQ 资源控制脚本就是此 OCF 脚本; 其二为 RabbitMQ 官方网站提供的资源代理 OCF 脚本 rabbitmq-server-ha^②, 要使用 rabbitmq-server-ha 资源代理脚本, 至少需要安装 rabbitmq-server3.6 以上版本。本节 OpenStack 高可用集群采用的 RabbitMQ 资源代理 OCF 脚本为官方推荐的 rabbitmq-server-ha, 安装的 RabbitMQ 软件包版本为 rabbitmq-server-3.6.0-1。此外, RabbitMQ 集群由三个控制节点构成, 消息的高可用通过队列在三个节点之间进行镜像来实现。RabbitMQ 资源在 Pacemaker 集群中的配置过程如下。

在全部控制节点上安装 RabbitMQ-Server-3.6 软件包:

① <https://github.com/ClusterLabs/resource-agents/blob/master/heartbeat/rabbitmq-cluster>。

② <https://www.rabbitmq.com/pacemaker.html>。

```
yum install -y rabbitmq-server
```

将每个控制节点的管理 IP 地址设置为 RabbitMQ 绑定的本地监听 IP 地址，设置命令参考如下：

```
echo "NODE_IP_ADDRESS=${bind_ip}" > /etc/rabbitmq/rabbitmq-env.conf
```

这里的变量 `${bind_ip}` 表示控制节点的管理 IP 地址，如对于 `controller1-vm`，则 `bind_ip` 变量值为 `192.168.142.110`。安装 `rabbitmq-server-3.6.0-1` 软件包后，在 RabbitMQ 的 OCF 脚本目录 `/usr/lib/ocf/resource.d/rabbitmq` 中会生成一个名为 `set_rabbitmq_policy.sh.example` 的参考脚本，该脚本内容主要用以配置 RabbitMQ 集群策略，其内容如下：

```
# This script is called by rabbitmq-server-ha.ocf during RabbitMQ
# cluster start up. It is a convenient place to set your cluster
# policy here, for example:
# ${OCF_RESKEY_ctl} set_policy ha-all "." '{"ha-mode":"all",\
"ha-sync-mode":"automatic"}' --apply-to all --priority 0
```

可以看到，`set_rabbitmq_policy.sh.example` 脚本的目的便是配置 RabbitMQ 集群的高可用策略，即队列镜像模式。该脚本由 RabbitMQ-Server 提供的 `rabbitmq-server-ha` 资源代理脚本自动调用，无须用户干预。为了进一步初始化符合 OpenStack 集群使用的 RabbitMQ 集群环境，这里将该参考脚本进行适当的自定义更改，使其在设置 RabbitMQ 集群策略的同时添加 OpenStack 客户端用户并设置用户权限。更改后的 `set_rabbitmq_policy.sh` 内容如下：

```
# This script is called by rabbitmq-server-ha.ocf during RabbitMQ
# cluster start up. It is a convenient place to set your cluster
# policy here, for example:
# ${OCF_RESKEY_ctl} set_policy ha-all "." '{"ha-mode":"all",\
"ha-sync-mode":"automatic"}' --apply-to all --priority 0

${OCF_RESKEY_ctl} set_policy ha-all "." '{"ha-mode":"all", \
"ha-sync-mode":"automatic"}' --apply-to all --priority 0
${OCF_RESKEY_ctl} add_user openstack openstack
${OCF_RESKEY_ctl} set_permissions openstack ".*" ".*" ".*"
```

更改 `set_rabbitmq_policy.sh` 脚本的执行权限：

```
chmod 755 /usr/lib/ocf/resource.d/rabbitmq/set_rabbitmq_policy.sh
```

现在即可在 Pacemaker 集群中创建 RabbitMQ 资源。根据 RabbitMQ 官方文档，使用的资源代理 RA 为 `ocf:rabbitmq:rabbitmq-server-ha`，RabbitMQ 资源以 Master/Slave 形式运行。资源创建过程如下：

```
pcs resource create rabbitmq-cluster ocf:rabbitmq:rabbitmq-server-ha\
--master erlang_cookie=DPMDALGUKEOMPTWHPYKC node_port=5672 \
op monitor interval=30 timeout=120 \
op monitor interval=27 role=Master timeout=120 \
op monitor interval=103 role=Slave timeout=120 OCF_CHECK_LEVEL=30 \
op start interval=0 timeout=120 \
```

```

op stop interval=0 timeout=120 \
op promote interval=0 timeout=60 \
op demote interval=0 timeout=60 \
op notify interval=0 timeout=60 \
meta notify=true ordered=false interleave=false master-max=1 \
master-node-max=1

```

资源创建完成之后, 检查 Pacemaker 集群中 RabbitMQ 多状态资源的运行情况, 正常情况下, 对于三节点的 RabbitMQ 集群, 应该存在一个 Master 节点和两个 Slave 节点。Pacemaker 集群资源信息如下:

```

[root@controller1-vm ~]# pcs resource
.....
Master/Slave Set: galera-master [galera]
  Masters: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: memcached-clone [memcached]
  Started: [ controller1-vm controller2-vm controller3-vm ]
Master/Slave Set: rabbitmq-cluster-master [rabbitmq-cluster]
  Masters: [ controller2-vm ]
  Slaves: [ controller1-vm controller3-vm ]

```

从 Pacemaker 集群的资源状态信息来看, RabbitMQ 集群 Master 节点为 controller2-vm, 更详细的集群信息可以通过 rabbitmqctl 命令来查看, 具体如下:

```

[root@controller1-vm ~]# rabbitmqctl cluster_status
Cluster status of node 'rabbit@controller1-vm' ...
[{nodes, [{disc, ['rabbit@controller1-vm', 'rabbit@controller2-vm',
                    'rabbit@controller3-vm']}]},
 {running_nodes, ['rabbit@controller3-vm', 'rabbit@controller2-vm',
                  'rabbit@controller1-vm']},
 {cluster_name, <<"rabbit@controller2-vm">>},
 {partitions, []}]

```

前文曾提及 rabbitmq-server-ha 资源代理会自动调用与其位于相同目录的集群策略设置脚本 set_rabbitmq_policy.sh。该脚本功能被自定义为设置 RabbitMQ 队列镜像、添加 RabbitMQ 用户 OpenStack 和设置该用户使用权限。RabbitMQ 集群策略是否已经如预期设置, 可以通过如下命令来验证:

```

//查看集群队列镜像策略
[root@controller1-vm ~]# rabbitmqctl list_policies
Listing policies ...
/   ha-all  all  .  {"ha-mode":"all","ha-sync-mode":"automatic"}  0
//查看集群用户
[root@controller1-vm ~]# rabbitmqctl list_users
Listing users ...
openstack      []
guest  [administrator]
//查看集群用户权限
[root@controller1-vm ~]# rabbitmqctl list_permissions
Listing permissions in vhost "/" ...

```

```

openstack      .*      .*      .*
quest          .*      .*      .*

```

可以看到, Pacemaker 管理下的 RabbitMQ 集群已经正常运行, 同时 RabbitMQ 集群已经按预期进行了初始化, 后续 OpenStack 相关组件可以直接使用 RabbitMQ 所提供的高可用消息队列服务。本节介绍的 RabbitMQ 高可用部署源代码可参考笔者位于 Github 上的开源项目 (<https://github.com/ynwssjx/Openstack-HA-Deployment>), 对应的部署脚本为 `5_create_rabbitmq_resource_on_pacemaker.sh`。

11.4.6 MongoDB 非关系数据库高可用部署

MongoDB 是使用较为广泛的非关系型数据库, 其在 OpenStack 集群中主要用于存储 Ceilometer 服务监控到的云平台参数信息, 即 MongoDB 的主要访问客户端为 OpenStack 的 Ceilometer 服务。关于 MongoDB 更多的工作原理及配置使用方法请参考第 7 章集群数据库系统相关的内容, 本节重点介绍 MongoDB 在 Pacemaker 集群中的高可用部署实现。MongoDB 集群的高可用通过其自带的 Replication Set 功能来实现, 在本节介绍的 OpenStack 高可用配置方案中, MongoDB 集群仍然部署在三个控制节点上。此外, MongoDB 以资源 Clone 的形式运行在三个控制节点上, 而高可用集群功能通过 MongoDB 自带的 Replication Set 来配置。要配置 MongoDB 资源, 首选需要在各个控制节点上安装 MongoDB 和 MongoDB-Server 软件包, 具体如下:

```
yum -y install mongodb mongodb-server
```

对 MongoDB 的配置文件 `/etc/mongodb.conf` 进行自定义设置, 包括指定 MongoDB 服务绑定的 IP 和 ReplSet 名称等, 具体如下:

```

echo "bind_ip = 0.0.0.0" >> /etc/mongod.conf
echo "replSet = ceilometer" >> /etc/mongod.conf
echo "smallfiles = true" >> /etc/mongod.conf

```

对 MongoDB 服务进行手动启停, 验证其可以正常启动, 具体如下:

```

systemctl start mongod.service
systemctl stop mongod.service

```

在 Pacemaker 集群中创建 MongoDB 资源, 资源以 Clone 形式运行在三个控制节点上, 资源创建命令如下:

```
pcs resource create mongodb systemd:mongod op start timeout=300s --clone
```

MongoDB 资源创建完成后, 在 Pacemaker 集群中可以查看 MongoDB 资源运行情况。正常情况下, MongoDB 应该同时运行在三个控制节点上, 具体如下:

```

[root@controller1-vm ~]# pcs resource
.....
Master/Slave Set: galera-master [galera]

```

```

Masters: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: memcached-clone [memcached]
Started: [ controller1-vm controller2-vm controller3-vm ]
Master/Slave Set: rabbitmq-cluster-master [rabbitmq-cluster]
Masters: [ controller3-vm ]
Slaves: [ controller1-vm controller2-vm ]
Clone Set: mongodb-clone [mongodb]
Started: [ controller1-vm controller2-vm controller3-vm ]

```

MongoDB 在三个控制节点正常运行后，即可进行 MongoDB 的 ReplSet 集群配置，其配置过程就是创建一个 ReplSet 集群并将三个控制节点分别加入集群中。这里 Repl Set 集群的名称已经在 /etc/mongod.conf 中配置为 ceilometer，此处仅需对 ReplSet 进行初始化并添加节点即可。具体过程参考如下代码段：

```

cat > /root/mongo_replica_setup.js << EOF
rs.initiate()
sleep(10000)
EOF
//为Repl Set集群添加节点
for node in $ha_node1 $ha_node2 $ha_node3; do
cat >> /root/mongo_replica_setup.js << EOF
rs.add("$node");
EOF
done
//执行mongo脚本
mongo /root/mongo_replica_setup.js

```

成功执行上述程序段后，MongoDB 的 ReplSet 集群便配置完成了。在 MongoDB 的 ReplSet 集群中，仅有一个 Primary 节点，其余节点均为 Secondary 节点。要查看 MongoDB 的 ReplSet 集群状态，可以进入 mongo 交互式命令行后运行 rs.status() 命令，具体如下：

```

[root@controller1-vm ~]# mongo
MongoDB shell version: 2.6.9
connecting to: test
Server has startup warnings:
.....
ceilometer:PRIMARY> rs.status()
{
  "set" : "ceilometer",
  "date" : ISODate("2016-11-30T07:09:08Z"),
  "myState" : 1,
  "members" : [
    {
      "_id" : 0,
      "name" : "controller1-vm:27017",
      "health" : 1,
      "state" : 1,
      "stateStr" : "PRIMARY",
      "uptime" : 458,

```



```

    "optime" : Timestamp(1480489714, 2),
    "optimeDate" : ISODate("2016-11-30T07:08:34Z"),
    "electionTime" : Timestamp(1480489704, 2),
    "electionDate" : ISODate("2016-11-30T07:08:24Z"),
    "self" : true
  },
  {
    "_id" : 1,
    "name" : "controller2-vm:27017",
    "health" : 1,
    "state" : 2,
    "stateStr" : "SECONDARY",
    "uptime" : 34,
    "optime" : Timestamp(1480489714, 2),
    "optimeDate" : ISODate("2016-11-30T07:08:34Z"),
    "lastHeartbeat" : ISODate("2016-11-30T07:09:08Z"),
    "lastHeartbeatRecv" : ISODate("2016-11-30T07:09:07Z"),
    "pingMs" : 5,
    "syncingTo" : "controller1-vm:27017"
  },
  {
    "_id" : 2,
    "name" : "controller3-vm:27017",
    "health" : 1,
    "state" : 2,
    "stateStr" : "SECONDARY",
    "uptime" : 34,
    "optime" : Timestamp(1480489714, 2),
    "optimeDate" : ISODate("2016-11-30T07:08:34Z"),
    "lastHeartbeat" : ISODate("2016-11-30T07:09:08Z"),
    "lastHeartbeatRecv" : ISODate("2016-11-30T07:09:08Z"),
    "pingMs" : 3,
    "syncingTo" : "controller1-vm:27017"
  }
],
"ok" : 1
}
ceilometer:PRIMARY>

```

可以看到，ReplSet 名称为 ceilometer，Primary 节点为 controller1-vm，而 controller2-vm 和 controller3-vm 均为 Secondary 节点。至此，MongoDB 的高可用集群已经配置完成，而且 MongoDB 服务由 Pacemaker 集群资源管理器自动管理，任何节点上的 Mongod 服务故障停止后，Pacemaker 都会自动重新启动该服务。同时，由于 MongoDB 集群通过 ReplSet 实现了节点之间的数据同步，因此任意节点上的 Mongd 服务故障均不会影响 OpenStack 集群中的 MongoDB 数据库服务。本节介绍的 MongoDB 高可用部署源代码可参考笔者位于 Github 上的开源项目（<https://github.com/ynwssjx/Openstack-HA-Deployment>），对应的部署脚本为 6_create_mongodb_resource_on_pacemaker。

11.5 本章小结

至此，OpenStack 集群中已经部署了高可用的 Pacemaker 集群、HAProxy 负载均衡软件、MariaDB 关系型数据库集群、Memcache 缓存系统、RabbitMQ 消息队列集群和 MongoDB 非关系型数据库集群。其中，Pacemaker 作为集群资源管理系统，负责管理集群中全部资源的启动、停止、重启和监控等行为，同时还负责对控制节点进程 Fencing 隔离操作。在此时的 Pacemaker 集群资源中，除了 Stonith 设备外，Pacemaker 管理的资源包括以 Clone 形式同时运行在控制节点集群上的 HAProxy 负载均衡资源、均匀分布在控制节点集群上的若干虚拟 IP 资源、以 Master/Slave 多主复制高可用集群形式运行的 MariaDB 关系型数据库资源、以 Clone 形式运行在控制节点集群上的 Memcache 缓存资源、以 Master/Slave 队列镜像形式运行在控制节点集群上的 RabbitMQ 消息队列资源和以 RepSet 高可用集群形式运行的 MongoDB 非关系型数据库资源。

Pacemaker 管理的上述资源是后续 OpenStack 高可用核心服务部署的关键和基础，同时也是后续 Pacemaker 集群中 OpenStack 相关服务资源可以正常启动的提前约束，仅有在这些基础服务正常启动并可以稳定提供服务之后，OpenStack 相关核心服务资源才能成功启动，因此本章内容是后续 OpenStack 服务高可用部署的基础。通过对本章内容的学习，读者应该可以部署基于 Pacemaker 集群的高可用负载均衡服务、虚拟 IP 服务、MariaDB 关系型数据库服务、Memcache 缓存服务、RabbitMQ 消息队列服务以及 MongoDB 非关系型数据库服务。下一章将重点介绍 Pacemaker 集群中 OpenStack 核心服务组件的高可用部署。

OpenStack 高可用集群核心服务部署

第 11 章对 OpenStack 所依赖的数据库和消息队列等相关基础软件进行了高可用部署实现，从而解决了 OpenStack 底层服务的高可用性，但是第 11 章并未涉及上层 OpenStack 核心服务的高可用实现，本章将在第 11 章的基础上重点介绍如何实现基于 Pacemaker 集群的 OpenStack 核心服务组件的高可用部署。OpenStack 开源云计算由众多独立开发，但又彼此交互的服务组件构成，为了实现 OpenStack 整体功能服务的高可用性，在生产环境中，构成 OpenStack 服务的每一个服务组件都必须实现高可用。在实际部署中，根据不同的功能需求，用户可以通过自定义部署服务组件来实现自己的 OpenStack 云计算，尤其是在“大帐篷”模式下，要一次性覆盖全部 OpenStack 项目似乎并不现实，对于普通云计算用户而言也是没有必要的，尤其是在私有云环境下，因此本章将重点介绍如何对 OpenStack 中最为成熟和核心、用户使用率最高的服务项目进行高可用部署，这些服务包括 Nova、Cinder、Neutron、Glance、Keystone、Ceilometer、Heat 和 Horizon，以上 OpenStack 项目覆盖了组成云计算最基本的计算、存储、网络、认证、测量、编排和 GUI 操作等服务。高可用部署完成后，本章将对 OpenStack 集群的高可用性进行验证，同时本章内容也是对前文所有知识点的综合应用实践，因此，通过本章所介绍的 OpenStack 高可用部署方案，读者不仅可以搭建满足生产环境要求的 OpenStack 高可用云计算环境，还将对 OpenStack 开源云计算的博大精深具有深入的体会和感悟。

12.1 OpenStack 控制节点服务高可用部署

本书将 OpenStack 高可用集群部署分为基础服务高可用部署和核心服务高可用部署，其中第 11 章对 OpenStack 高可用集群所依赖的数据库、消息队列和负载均衡等基础

服务进行了高可用部署介绍和实现，本章将对 OpenStack 高可用集群的核心服务进行高可用部署介绍，本节所要实现的 OpenStack 高可用部署服务包括计算服务 Nova、存储服务 Cinder、网络服务 Neutron、认证授权服务 Keystone、镜像服务 Glance、数据采集监控服务 Ceilometer、编排服务 Heat 和控制面板服务 Horizon，几乎覆盖了主流 OpenStack 部署中所要实现的服务项目。必须指出的是，以上 OpenStack 服务项目的部署依赖于数据库和消息队列等基础服务，因此在进行本节所介绍的 OpenStack 高可用集群核心服务的高可用部署之前，本书第 11 章中介绍的基础高可用服务必须在 Pacemaker 集群中正常运行。

12.1.1 Keystone 认证服务高可用部署

Keystone 为 OpenStack 云平台提供了身份验证、授权和服务目录管理等集成身份认证服务。在 OpenStack 的诸多服务中，用户首选需要与其交互的便是 Keystone 服务，一旦用户被 Keystone 认证通过，该用户便可利用 Keystone 颁发的令牌访问其他 OpenStack 服务。此外，OpenStack 中的其他服务也需要借用 Keystone 的认证功能来确定访问用户的身份，并通过 Keystone 发现其他 OpenStack 服务的访问地址。OpenStack 用户和服务通过 Keystone 的认证管理和服务编目功能发现其他 OpenStack 服务的访问地址，服务编目是 OpenStack 中全部部署服务的集合，当用户在 OpenStack 环境中安装部署新的服务项目时，便会将该服务注册到 Keystone 中，而每个服务均有一个或多个访问地址（Endpoint），每个 Endpoint 有三种类型，即 Admin、Public 和 Internal。在生产环境中，为了安全起见，通常将不同类型的 Endpoint（不同网段的 IP 地址）分配给不同类型的用户使用，例如 Public 类型的 Endpoint 可以从 Internet 访问，以便用户可以通过 Internet 访问并管理他们的云计算资源，而 Admin 类型的 Endpoint 被限制在企业内网内，仅云平台操作人员可以访问。Keystone 在 OpenStack 开源云计算平台概念架构中的位置如图 12-1 所示。

由于 Keystone 是 OpenStack 云平台操作和运维的基础和前提，Keystone 不仅负责 OpenStack 用户身份授权认证与管理，还通过服务访问地址的编目功能向用户和 OpenStack 服务提供全部 OpenStack 服务的访问地址。因此，Keystone 在 OpenStack 集群中的位置举足轻重，失去了 Keystone 的服务，OpenStack 集群将失去全部对外服务的功能。对于生产环境而言，Keystone 服务的高可用部署是必须的，本节将重点介绍如何在 Pacemaker 集群中部署高可用的 Keystone 服务，在部署 Keystone 之前，必须确保数据库和消息队列等基础服务已经在 Pacemaker 集群中正常运行。在本节介绍的 Keystone 高可用部署中，Keystone 服务的高可用通过 HAProxy 负载均衡来实现，即 Pacemaker 集群中的三个控制节点以 Clone 资源的形式同时运行 Keystone 服务，客户端通过虚拟 IP 地址对 Keystone 服务的访问请求被 HAProxy 以一定的负载均衡算法转发到后端运行 Keystone 服务的三个控制节点上，在这个过程中，用户的每一次请求仅会被转发至后端的一个控制节点，而任何控制节点的故障均不会影响到其他控制节点对 Keystone 服务的请求处理，除非三个控制节点全部故障，否则 OpenStack 高可用集群的 Keystone 服务将不会存在单点故障。Keystone 资源在

Pacemaker 集群中的安装配置过程如下：

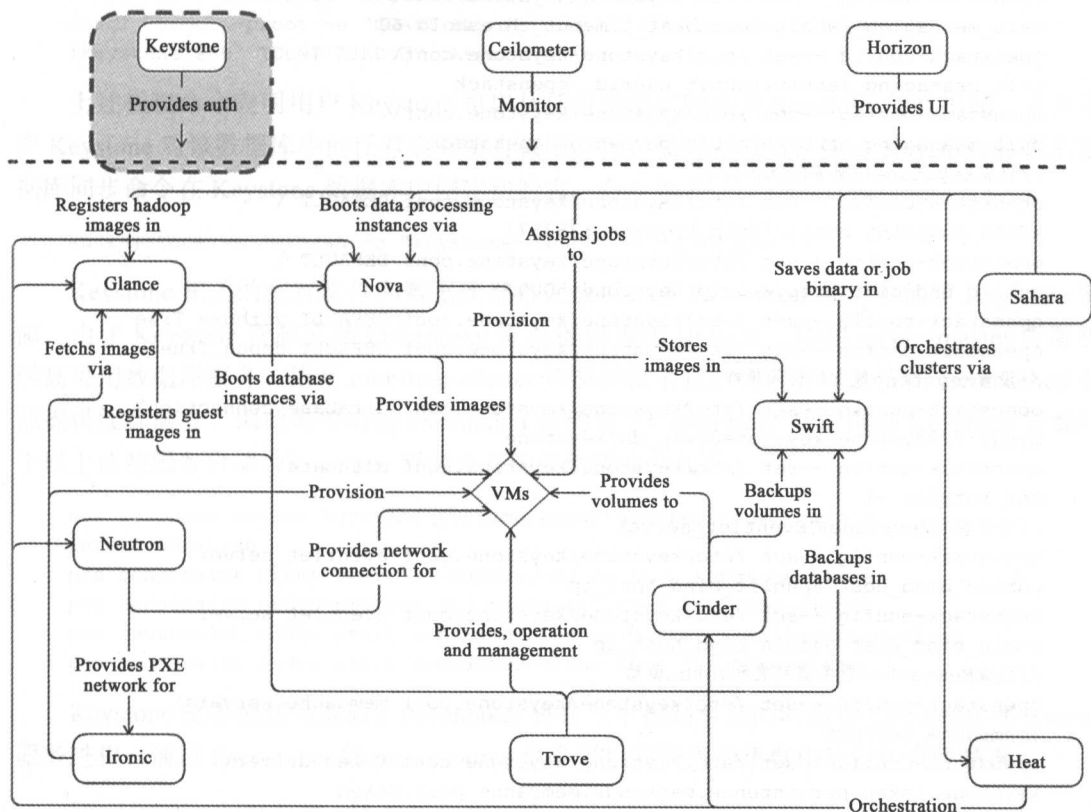


图 12-1 Keystone 在 OpenStack 概念架构中的位置

在 Pacemaker 控制节点集群安装 Keystone 相关软件包：

```
yum install -y openstack-keystone openstack-utils \
python-openstackclient python-keystoneclient
```

Keystone 服务可以通过 Eventlet 或者 WSGI 应用 HTTP 的形式运行在 Linux 系统中，采用不同的运行方式则需要安装不同的软件包，这里将 Keystone 以 Eventlet 形式运行。Keystone 安装完成后，在启动 Keystone 相关服务进程之前，需要配置 Keystone，配置文件为 /etc/keystone/keystone.conf，配置文件中的主要配置内容包括 RabbitMQ 服务访问策略、MariaDB 访问策略、Memcached 服务器列表以及 Token 驱动等，配置命令参考如下：

```
//设置初始化时使用的Token
```

```
openstack-config --set /etc/keystone/keystone.conf DEFAULT admin_token\
$keystone_secret
```

```
//设置rabbitmq访问策略
```

```
openstack-config --set /etc/keystone/keystone.conf\
oslo_messaging_rabbit rabbit_hosts $rabbitmq_hosts
openstack-config --set /etc/keystone/keystone.conf\
```



```

oslo_messaging_rabbit rabbit_ha_queues true
openstack-config --set /etc/keystone/keystone.conf\
oslo_messaging_rabbit heartbeat_timeout_threshold 60
openstack-config --set /etc/keystone/keystone.conf\
oslo_messaging_rabbit rabbit_userid openstack
openstack-config --set /etc/keystone/keystone.conf\
oslo_messaging_rabbit rabbit_password openstack
//设置keystone的服务Endpoint
openstack-config --set /etc/keystone/keystone.conf DEFAULT \
admin_endpoint http://$vip_keystone:35357/
openstack-config --set /etc/keystone/keystone.conf DEFAULT \
public_endpoint http://$vip_keystone:5000/
openstack-config --set /etc/keystone/keystone.conf DEFAULT verbose True
openstack-config --set /etc/keystone/keystone.conf DEFAULT debug True
//设置Keystone数据库访问策略
openstack-config --set /etc/keystone/keystone.conf database connection\
mysql://keystone:keystone@$vip_db/keystone
openstack-config --set /etc/keystone/keystone.conf database\
max_retries -1
//设置运行Keystone的Eventlet_server
openstack-config --set /etc/keystone/keystone.conf eventlet_server
public_bind_host $public_bind_host_ip
openstack-config --set /etc/keystone/keystone.conf eventlet_server
admin_bind_host $admin_bind_host_ip
//设置Memcached服务器列表和Token驱动
openstack-config --set /etc/keystone/keystone.conf memcache servers\
$memcache_servers
openstack-config --set /etc/keystone/keystone.conf token driver\
keystone.token.persistence.backends.memcache_pool.Token
openstack-config --set /etc/keystone/keystone.conf token provider\
keystone.token.providers.uuid.Provider
openstack-config --set /etc/keystone/keystone.conf revoke driver \
keystone.contrib.revoke.backends.sql.Revoke

```

配置过程中有几个地方需要注意，Memcache 服务器列表由三个运行 Memcached 服务的控制节点组成，因为这里采用 Eventlet_Server 来运行 Keystone，所以建议使用的 Token 驱动为 `keystone.token.persistence.backends.memcache_pool.Token`，如果使用 Apache 与 Mod_WSGI 组合形式来运行 Keystone，则建议选择 `keystone.token.persistence.backends.memcache.Token` 驱动，而在 OpenStack 的新版本 Mitaka 和 Newton 中，已经不再需要对 Token 进行持久化存储，Keystone 的 Token 由 Fernet 来提供。

Keystone 配置完成后，不要手动启动 Keystone 服务或对其设置开机自启动，Keystone 服务进程的相关控制交由 Pacemaker 完成。在第 11 章中安装配置高可用 MariaDB 数据库时，已经创建了名为 Keystone 的数据库，而在 Keystone 的配置文件中，使用的是 Keystone 这个用户名来访问该数据库，因此需要对用户 Keystone 进行数据库相关的授权，数据库授权命令如下：

```
mysql -uroot -proot -e "GRANT ALL PRIVILEGES ON keystone.* TO\
'keystone'@'localhost' IDENTIFIED BY 'keystone';"
mysql -uroot -proot -e "GRANT ALL PRIVILEGES ON keystone.* TO \
'keystone'@'%' IDENTIFIED BY 'keystone';"
```

上述授权命令表明用户 Keystone 可以从任意主机访问名为 Keystone 的数据库，并且用户 Keystone 对该数据库中的任意表有所有权限。用户授权完成后，通过 Keystone 提供的数据库同步命令在 Keystone 数据库中创建初始表，命令如下：

```
su keystone -s /bin/sh -c "keystone-manage -v -d db_sync"
```

Keystone 相关的配置文件和数据库准备完成后，在 Pacemaker 集群中创建 Keystone 资源，由于 Keystone 资源依赖 lb-haproxy 资源提供虚拟 IP 服务、依赖 galera-master 资源提供高可用数据库服务、依赖 rabbitmq-cluster 资源提供消息队列服务以及依赖 memcached 资源提供缓存服务，因此需要通过 Pacemaker 的约束（Constraint）功能限制 Keystone 资源后于以上这些服务启动，Keystone 资源及其约束创建命令如下：

```
pcs resource create keystone systemd:openstack-keystone --clone \
interleave=true
pcs constraint order start lb-haproxy-clone then keystone-clone
pcs constraint order promote galera-master then keystone-clone
pcs constraint order start rabbitmq-cluster-master then keystone-clone
pcs constraint order start memcached-clone then keystone-clone
```

Keystone 资源创建完成后，Pacemaker 将在三个控制节点同时启动 Openstack-Keystone 服务进程，通过 Pacemaker 的 pcs resource 命令可以看到当前集群中运行的资源情况：

```
[root@controller3-vm ~]# pcs resource
.....
Master/Slave Set: galera-master [galera]
  Masters: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: memcached-clone [memcached]
  Started: [ controller1-vm controller2-vm controller3-vm ]
Master/Slave Set: rabbitmq-cluster-master [rabbitmq-cluster]
  Masters: [ controller1-vm ]
  Slaves: [ controller2-vm controller3-vm ]
Clone Set: mongodb-clone [mongodb]
  Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: keystone-clone [keystone]
  Started: [ controller1-vm controller2-vm controller3-vm ]
```

从 Pacemaker 的资源运行情况中可以看到，Keystone 资源已经同时运行在三个控制节点上，现在可以通过预先置入配置文件中的 Token^①、Keystone 的访问地址以及默认的 Region 来访问 Keystone 服务，为了简单起见，将以上变量值导入系统环境变量：

```
export OS_TOKEN=$keystone_secret
```

^① 为了安全起见，在 Keystone 初始化完成后，建议删除配置文件中的内置 Token。

```
export OS_URL="http://$vip_keystone:35357/v2.0"
export OS_REGION_NAME=regionOne
```

下面即可通过内置 Token 访问 Keystone 服务，并为 Keystone 创建 Service 和 Endpoint，同时为 OpenStack 集群创建一个管理员用户和一个普通测试用户，具体命令如下：

```
//创建service
openstack service create --name=keystone --description="Keystone \
Identity Service" identity
//创建endpoint
openstack endpoint create --publicurl "http://$vip_keystone:5000/v2.0"
--adminurl "http://$vip_keystone:35357/v2.0" \
--internalurl "http://$vip_keystone:5000/v2.0" \
--region regionOne keystone
//创建管理员用户
openstack user create --password admin admin
openstack role create admin
openstack project create admin
openstack role add --project admin --user admin admin
//创建普通测试用户
openstack user create --password demo demo
openstack role create _member_
openstack project create demo
openstack role add --project demo --user demo _member_
```

为了便于管理员用户和测试用户访问 OpenStack 的 Keystone 服务，可以将访问 Keystone 服务所需的系统环境变量集中到一个文件中，通常将其称为 rc 文件，用户可以通过 source 不同类型的 rc 文件来以不同的角色访问 Keystone 服务，管理员用户和普通测试用户的 rc 文件的生成过程参考如下：

```
//管理员用户rc文件(adminrc)
cat > /root/adminrc << EOF
export OS_USERNAME=admin
export OS_TENANT_NAME=admin
export OS_PROJECT_NAME=admin
export OS_REGION_NAME=regionOne
export OS_PASSWORD=admin
export OS_AUTH_URL=http://$vip_keystone:35357/v2.0/
export PS1='[\u@\h \W(keystone_admin)]\$ '
EOF
//测试用户rc文件(demorc)
cat > /root/demorc << EOF
export OS_USERNAME=demo
export OS_TENANT_NAME=demo
export OS_PROJECT_NAME=demo
export OS_REGION_NAME=regionOne
export OS_PASSWORD=demo
export OS_AUTH_URL=http://$vip_keystone:5000/v2.0/
export PS1='[\u@\h \W(keystone_user)]\$ '
EOF
```

Keystone 是 OpenStack 集群全部用户和服务的集中管理者，为了后续安装部署方便，此处将本章需要部署的全部 OpenStack 服务项目所涉及的用户、服务以及 Endpoint 进行集中创建，后续在安装配置各个服务项目时无须再创建与之相关的用户及服务，OpenStack 服务项目的集中创建命令参考如下：

```
//创建一个针对Openstack内部服务项目的Project
openstack project create --description "Services Tenant" services
//为Glance服务创建用户、服务及endpoint
openstack user create --password glance glance
openstack role add --project services --user glance admin
openstack service create --name=glance --description="Glance Image
Service" image
openstack endpoint create \
--publicurl "http://$vip_glance:9292" \
--adminurl "http://$vip_glance:9292" \
--internalurl "http://$vip_glance:9292" \
--region regionOne glance
//为Cinder服务创建用户、服务及endpoint
openstack user create --password cinder cinder
openstack role add --project services --user cinder admin
openstack service create --name=cinder --description="Cinder Volume
Service" volume
openstack endpoint create \
--publicurl "http://$vip_cinder:8776/v1/\$(tenant_id)s" \
--adminurl "http://$vip_cinder:8776/v1/\$(tenant_id)s" \
--internalurl "http://$vip_cinder:8776/v1/\$(tenant_id)s" \
--region regionOne cinder
openstack service create --name=cinderv2 --description="OpenStack Block
Storage" volumev2
openstack endpoint create \
--publicurl "http://$vip_cinder:8776/v2/\$(tenant_id)s" \
--adminurl "http://$vip_cinder:8776/v2/\$(tenant_id)s" \
--internalurl "http://$vip_cinder:8776/v2/\$(tenant_id)s" \
--region regionOne cinderv2
//为Neutron服务创建用户、服务及endpoint
openstack user create --password neutron neutron
openstack role add --project services --user neutron admin
openstack service create --name=neutron --description="OpenStack
Networking Service" network
openstack endpoint create \
--publicurl "http://$vip_neutron:9696" \
--adminurl "http://$vip_neutron:9696" \
--internalurl "http://$vip_neutron:9696" \
--region regionOne neutron
//为Nova服务创建用户、服务及endpoint
openstack user create --password nova nova
openstack role add --project services --user nova admin
openstack service create --name=compute --description="OpenStack
```

```

Compute Service" compute
openstack endpoint create \
  --publicurl "http://$vip_nova:8774/v2/\$(tenant_id)s" \
  --adminurl "http://$vip_nova:8774/v2/\$(tenant_id)s" \
  --internalurl "http://$vip_nova:8774/v2/\$(tenant_id)s" \
  --region regionOne compute
//为Heat服务创建用户、服务及endpoint
openstack user create --password heat heat
openstack role add --project services --user heat admin
openstack service create --name=heat --description="Heat Orchestration
Service" orchestration
openstack endpoint create \
  --publicurl "http://$vip_heat:8004/v1/\$(tenant_id)s" \
  --adminurl "http://$vip_heat:8004/v1/\$(tenant_id)s" \
  --internalurl "http://$vip_heat:8004/v1/\$(tenant_id)s" \
  --region regionOne heat
openstack service create --name=heat-cfn --description="Heat
CloudFormation Service" cloudformation
openstack endpoint create \
  --publicurl "http://$vip_heat:8000/v1" \
  --adminurl "http://$vip_heat:8000/v1" \
  --internalurl "http://$vip_heat:8000/v1" \
  --region regionOne heat-cfn
//为Ceilometer服务创建用户、服务及endpoint
openstack user create --password ceilometer ceilometer
openstack role add --project services --user ceilometer admin
openstack role create ResellerAdmin
openstack role add --project services --user ceilometer ResellerAdmin
openstack service create --name=ceilometer --description="OpenStack
Telemetry Service" metering
openstack endpoint create \
  --publicurl "http://$vip_ceilometer:8777" \
  --adminurl "http://$vip_ceilometer:8777" \
  --internalurl "http://$vip_ceilometer:8777" \
  --region regionOne ceilometer

```

OpenStack 全部服务项目的用户服务及 Endpoint 创建完成后，通过 OpenStack 客户端命令行即可查看 Keystone 所管理的 User、Service 和 Project 等信息，如下：

```

[root@controller1-vm ~]# source adminrc
[root@controller1-vm ~(keystone_admin)]$ openstack service list
+-----+-----+-----+
| ID                                | Name          | Type          |
+-----+-----+-----+
| 16c4d7c0c8534339b539ea1521e12642 | compute       | compute       |
| 30316f9b653444f7b574bb075d79391a | ceilometer    | metering      |
| 3b69e01041bf4d969fe104ec4b7bc4b0 | cinderv2      | volumev2      |
| 6db7261e2bfc40049a859433cf9c6c1c | keystone      | identity      |
| 921eed646838484197285c23ffa3a2ad | cinder        | volume        |
| b95d049d2feb43e888a2383a5c59b48a | heat          | orchestration |

```

```
| da23e029762c492c8a0bc7c0bcebelca | neutron | network |
| f394fbb3c37f4e9da8d5e1dded40f30b | heat-cfn | cloudformation |
| fa80fb7850ae464294eb577722b8849a | glance | image |
```

```
[root@controller1-vm ~(keystone_admin)]$ openstack project list
```

```
+-----+-----+
| ID | Name |
+-----+-----+
| 224ef153cdb44985b2634dbd0a3d5135 | admin |
| ae7ccd4cdf2647989ad7544fceaa4945 | services |
| bc8e855e47544906aa85c281feed719f | demo |
```

```
[root@controller1-vm ~(keystone_admin)]$ openstack user list
```

```
+-----+-----+
| ID | Name |
+-----+-----+
| 0ee719bcd6844efb98c15e6f8631e0db | glance |
| 24884539b49e4134ace4c6749926518f | admin |
| 66fb0420ff1345a08fa34b7197bed392 | neutron |
| 8ff83b9d496547069957b99059384672 | heat |
| a0dacecd55b949c299de43fc572b16b5 | cinder |
| c8c122a5e9674adda30123773c74f5c0 | demo |
| d0ea47fed596474c96a73958ec588dcc | nova |
| eac955b4246048479f0d7e0744288657 | ceilometer |
```

截至目前，Keystone 高可用服务已经部署完成，同时 Keystone 已将 Openstack 各个服务项目进行了编目汇聚，各服务的访问地址、服务名称和用户等相关信息均已在 Keystone 中进行了注册，即 Keystone 所需要管理的全部 OpenStack 集群信息已经完全准备就绪，后续各个 OpenStack 服务项目在安装部署过程中只需访问 Keystone 即可获取所需访问目标项目的访问地址，并对其进行认证授权访问。本节介绍的 Keystone 高可用部署源代码可参考笔者位于 Github 上的开源项目（<https://github.com/ynwssjx/Openstack-HA-Deployment>），具体的部署脚本为 `7_create_keystone_resource_on_pacemaker.sh`，执行该脚本后，将会在 Pacemaker 集群中创建高可用的 Keystone 服务。

12.1.2 Glance 镜像服务高可用部署

Glance 镜像服务是 OpenStack 云平台的核心服务之一，Glance 接受客户端对磁盘或者服务器镜像的访问请求，以及来自终端用户和计算组件对镜像元数据的定义。此外，Glance 还支持将磁盘或服务器镜像存储在不同类型的存储库中，如可以将 Glance 镜像存储在 OpenStack 的对象存储服务 Swift、本地 / 网络文件系统、Ceph RADOS 块存储以及 AWS S3 存储中，Glance 镜像服务在 OpenStack 开源云计算平台概念架构中的位置以及与其他项目的交互关系如图 12-2 所示。

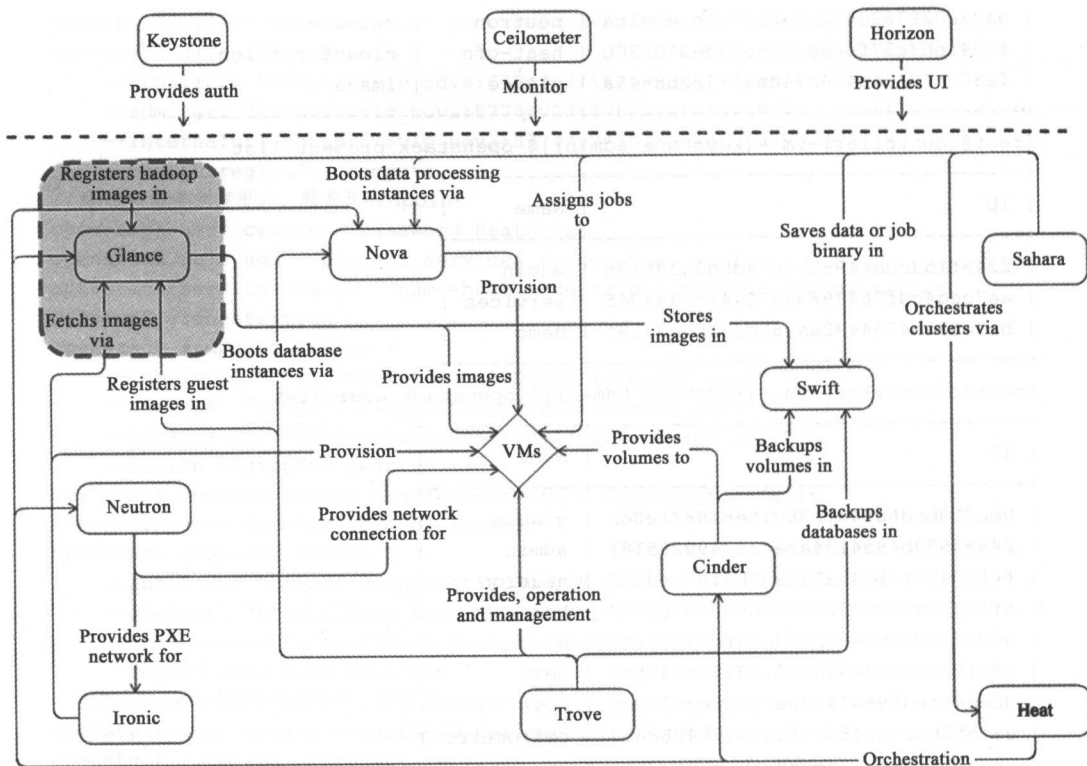


图 12-2 Glance 在 OpenStack 概念架构中的位置

在生产环境中，Glance 服务必须实现高可用部署，否则与镜像相关的全部功能将无法正常进行，如创建虚拟机过程将会因为不能访问镜像服务而无法创建。由于 Glance 服务分为 Glance-api 和 Glance-registry 两个部分，同时 Glance 服务还涉及存储部分，因此 Glance 服务项目的高可用部署不能如 Keystone 一样简单地将其同时运行在三个控制节点上，并通过 HAProxy 的负载均衡功能来实现。在 Pacemaker 集群中，如果使用文件系统来作为 Glance 的存储后端，则首先需要创建一个 Filesystem 类型的资源，而如果使用如 Ceph RADOS 块存储设备这种本身具备高可用性的存储集群，则只需创建 Glance-api 和 Glance-registry 资源即可。为了适用和简单起见，本节使用 NFS 文件系统作为 Glance 存储后端，这里将 VMware 虚拟宿主机 controller1 配置为 NFS 服务器，并在 Pacemaker 集群中创建 NFS 类型的文件系统资源，将 NFS 服务器导出的 Glance 镜像存储目录同时挂载在三个 OpenStack 集群控制节点上，然后再在 Pacemaker 集群中创建 Glance-api 和 Glance-registry 资源，并将 Glance-api 和 Glance-registry 以 Clone 资源的形式运行在三个控制节点上，通过 HAProxy 实现其服务的高可用。Glance 服务在 Pacemaker 集群中的高可用部署过程如下。

安装配置与 Glance 相关的软件包：

```
yum install -y openstack-glance openstack-utils python-openstackclient
```

安装完成后,对 Glance 进行配置,这里需要注意 Glance 的配置分为 Glance-api 的配置和 Glance-registry 的配置,配置文件分别为 /etc/glance/glance-api.conf 和 /etc/glance/glance-registry.conf, Glance 的配置过程参考如下:

```
//Glance-api服务配置
openstack-config --set /etc/glance/glance-api.conf database connection\
mysql://glance:glance@$vip_db/glance
openstack-config --set /etc/glance/glance-api.conf \
database max_retries -1
openstack-config --set /etc/glance/glance-api.conf paste_deploy flavor\
keystone
openstack-config --set /etc/glance/glance-api.conf keystone_authtoken\
identity_uri http://$vip_keystone:35357/
openstack-config --set /etc/glance/glance-api.conf keystone_authtoken\
auth_uri http://$vip_keystone:5000/
openstack-config --set /etc/glance/glance-api.conf keystone_authtoken\
admin_tenant_name services
openstack-config --set /etc/glance/glance-api.conf keystone_authtoken\
admin_user glance
openstack-config --set /etc/glance/glance-api.conf keystone_authtoken\
admin_password glance
openstack-config --set /etc/glance/glance-api.conf DEFAULT \
notification_driver messaging
openstack-config --set /etc/glance/glance-api.conf \
oslo_messaging_rabbit rabbit_userid openstack
openstack-config --set /etc/glance/glance-api.conf \
oslo_messaging_rabbit rabbit_password openstack
openstack-config --set /etc/glance/glance-api.conf \
oslo_messaging_rabbit rabbit_hosts $rabbitmq_hosts
openstack-config --set /etc/glance/glance-api.conf \
oslo_messaging_rabbit rabbit_ha_queues true
openstack-config --set /etc/glance/glance-api.conf \
oslo_messaging_rabbit heartbeat_timeout_threshold 60
openstack-config --set /etc/glance/glance-api.conf DEFAULT \
registry_host $vip_glance
openstack-config --set /etc/glance/glance-api.conf DEFAULT bind_host\
$public_bind_host_ip
openstack-config --set /etc/glance/glance-api.conf DEFAULT verbose \
True
openstack-config --set /etc/glance/glance-api.conf DEFAULT debug \
True
//配置Glance-registry服务
openstack-config --set /etc/glance/glance-registry.conf database\
connection mysql://glance:glance@$vip_db/glance
openstack-config --set /etc/glance/glance-registry.conf database \
max_retries -1
openstack-config --set /etc/glance/glance-registry.conf paste_deploy\
flavor keystone
openstack-config --set /etc/glance/glance-registry.conf \
keystone_authtoken identity_uri http://$vip_keystone:35357/
```

```

openstack-config --set /etc/glance/glance-registry.conf \
keystone_authtoken auth_uri http://$vip_keystone:5000/
openstack-config --set /etc/glance/glance-registry.conf \
keystone_authtoken admin_tenant_name services
openstack-config --set /etc/glance/glance-registry.conf \
keystone_authtoken admin_user glance
openstack-config --set /etc/glance/glance-registry.conf \
keystone_authtoken admin_password glance
openstack-config --set /etc/glance/glance-registry.conf DEFAULT \
notification_driver messaging
openstack-config --set /etc/glance/glance-registry.conf DEFAULT \
verbose True
openstack-config --set /etc/glance/glance-registry.conf DEFAULT \
debug True
openstack-config --set /etc/glance/glance-registry.conf \
oslo_messaging_rabbit rabbit_userid openstack
openstack-config --set /etc/glance/glance-registry.conf \
oslo_messaging_rabbit rabbit_password openstack
openstack-config --set /etc/glance/glance-registry.conf \
oslo_messaging_rabbit rabbit_hosts $rabbitmq_hosts
openstack-config --set /etc/glance/glance-registry.conf \
oslo_messaging_rabbit rabbit_ha_queues true
openstack-config --set /etc/glance/glance-registry.conf \
oslo_messaging_rabbit heartbeat_timeout_threshold 60
openstack-config --set /etc/glance/glance-registry.conf DEFAULT \
registry_host $vip_glance
openstack-config --set /etc/glance/glance-registry.conf DEFAULT \
bind_host $public_bind_host_ip

```

因为采用 NFS 文件系统作为 Glance 的后端存储，因此需要在 NFS 服务器上（controller1）创建一个共享目录，如下：

```
mkdir -p /data/glance
```

在 Pacemaker 集群中创建 NFS 文件系统资源，文件系统资源将会把 NFS 服务器上的 /data/glance 目录挂载到运行 glance-fs 资源的本地节点 /var/lib/glance 目录上，文件系统资源的创建命令参考如下：

```

//这里的$master表示NFS Server主机名
pcs resource create glance-fs Filesystem device="$master:/data/glance" \
directory="/var/lib/glance" fstype="nfs" options="v3" --clone

```

待 glance-fs 资源在三个控制节点上稳定运行后，检查控制节点上 Pacemaker 是否已经通过 NFS 将 /data/glance 挂载到本地 /var/lib/glance 目录上，如下：

```

[root@controller2-vm ~]# df -h
Filesystem                Size  Used Avail Use% Mounted on
.....
controller1:/data/glance  36G   25G   12G   69% /var/lib/glance

```

之后, 将本地 `/var/lib/glance` 目录设置为对 `glance` 用户可读写, 并同步 Glance 数据库, 在 `glance` 数据库中创建初始表, 如下:

```
chown glance:nobody /var/lib/glance
su glance -s /bin/sh -c "glance-manage db_sync"
```

Glance 服务启动所需的配置文件和数据库准备工作完成后, 便可在 Pacemaker 集群中创建 Glance-api 和 Glance-registry 资源, 由于 Glance-api 和 Glance-registry 均依赖 Glance-fs 资源, 因此必须通过 Pacemaker 的绑定约束 (Colocation) 将 Glance-api 和 Glance-registry 资源与 Glance-fs 绑定到一个节点上运行, 并按照 Glance-fs->Glance-api->Glance-registry 的约束顺序启动资源, 同时由于 Glance 资源要访问 Keystone, 因此 Keystone 资源必须在 Glance 之前启动, Pacemaker 资源创建过程如下:

```
pcs resource create glance-registry systemd:openstack-glance-registry\
--clone interleave=true
pcs resource create glance-api systemd:openstack-glance-api --clone\
interleave=true
//创建约束
pcs constraint order start glance-fs-clone then glance-registry-clone
pcs constraint colocation add glance-registry-clone with glance-fs-clone
pcs constraint order start glance-registry-clone then glance-api-clone
pcs constraint colocation add glance-api-clone with \
glance-registry-clone
pcs constraint order start keystone-clone then glance-registry-clone
```

Glance-api 和 Glance-registry 资源创建完成后, Pacemaker 将在三个控制节点上同时启动 Glance-api 和 Glance-registry 资源, 通过 Pacemaker 集群的 `pcs resource` 命令可以查看当前 Pacemaker 集群中的资源运行情况, 如下:

```
[root@controller1-vm ~]# pcs resource
.....
Clone Set: keystone-clone [keystone]
    Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: glance-fs-clone [glance-fs]
    Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: glance-registry-clone [glance-registry]
    Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: glance-api-clone [glance-api]
    Started: [ controller1-vm controller2-vm controller3-vm ]
```

可以看到, `glance-fs`、`glance-registry` 和 `glance-api` 资源已经按照先后顺序全部启动在 Pacemaker 集群中的三个控制节点上。为了验证 Glance 的功能是否正常, 可以制作一个测试镜像并对其查询来进行验证, 采用裸容器格式制作一个 `qcow2` 类型的公有镜像, 并观察镜像制作进度, 其命令参考如下:

```
glance image-create --container-format bare --disk-format qcow2 \
--is-public true --file cirros-0.3.4-x86_64-disk.img --name cirros \
```

```
--progress
```

查询 Glance 镜像，结果如下：

```
[root@controller1-vm ~(keystone_admin)]$ glance image-list
```

ID	Name	Disk Format	Container Format	Size	Status
...e9c018	cirros	qcow2	bare	13287936	active

至此，高可用 Glance 服务已经部署完成，并且 Glance 服务已经能够提供正常的镜像服务。Glance 项目的 API 和 Registry 服务被分别以 Pacemaker Clone 资源的形式同时运行在三个控制节点上，并通过 HAProxy 实现了负载均衡高可用，因此在 Openstack 集群中，Glance 服务将不存在单点故障问题，后续将继续对其他 OpenStack 服务进行高可用部署。本节介绍的 Glance 高可用部署源代码可参考笔者位于 Github 上的开源项目（<https://github.com/ynwssjx/Openstack-HA-Deployment>），具体的部署脚本为 8_create_glance_resource_on_pacemaker.sh，执行该脚本后，将会在 Pacemaker 集群中创建高可用的 Glance 服务。

12.1.3 Cinder 块存储服务高可用部署

Cinder 块存储服务是 OpenStack 核心服务之一，其主要负责为 OpenStack 集群提供持久化的块存储服务。在 OpenStack 实际应用中，Cinder 负责向客户虚拟机提供块存储设备，Cinder 对存储的提供和使用方式由用户所采用的存储驱动决定，而在多后端配置中则由多个后端块存储驱动各自决定。Cinder 支持各种开源或商业的存储驱动，常见的驱动包括 NAS/SAN、NFS、iSCSI、LVM 和 Ceph RBD 等。在 OpenStack 集群中，Cinder 提供了最基本的存储卷管理功能，同时 Cinder 与 OpenStack 的计算服务 Nova 交互从而为其虚拟机提供块存储设备，除此之外，Cinder 还提供了存储卷快照管理、存储卷类型管理和存储卷备份管理等功能。Cinder 主要由 Cinder-api、Cinder-scheduler 和 Cinder-volume 服务组成，对于正常的 OpenStack 集群部署和存储服务功能，上述三个 Cinder 服务组件必须部署，而根据用户需求，可以选择性地部署 Cinder-backup 服务。通常 Cinder-api 和 Cinder-scheduler 部署在 OpenStack 集群控制节点上，而 Cinder-volume 则根据用户所选用的 Volume 驱动可以部署在控制节点、计算节点和独立的存储节点上。图 12-3 为 Cinder 在 OpenStack 开源云计算概念架构设计中的位置及其与其他 OpenStack 服务项目的交互。

在 Cinder 的三个服务组件 Cinder-api、Cinder-scheduler 和 Cinder-volume 中，Cinder-api 和 Cinder-scheduler 的高可用问题实现较为简单，只需将其以 Clone 资源的形式运行在 Pacemaker 集群中的三个控制节点上，并通过 HAProxy 的负载均衡来实现其高可用即可，但是 Cinder-volume 的高可用一直没有得到理想的解决，不管是 Redhat，还是 Mirantis 的 OpenStack 高可用方案，Cinder-volume 服务的 Active/Active 高可用模式均存在可能的数据

损坏风险，因此 Redhat 在其 OSP 架构中已经将 Cinder-volume 服务改为 Active/Passive 模式^①。而对于 Cinder-volume 驱动后端底层存储设备的高可用，业界有很多不同的实现方式，如传统的磁盘阵列冗余技术（RAID）、IBM SVC/EMC VPLEX 存储阵列实时同步技术以及开源存储集群 Ceph 等，均可以在不同程度上实现底层存储设备的高可用。在正式生产环境中，如果条件允许，使用 SVC/VPLEX 商业解决方案所提供的 Cinder-volume 驱动是比较理想的选择，而随着 Ceph 分布式存储集群的发展和成熟，采用 Ceph RBD 的后端驱动也可以实现高可用的存储服务。基于简单适用的原则，本节采用 NFS 驱动作为 Cinder-volume 的后端，同时 Cinder-volume 以 Active/Passive 的形式运行在 OpenStack 控制节点上，节点服务由 Passive 到 Active 之间的转换由 Pacemaker 来控制，由于不能实现 Active/Active 模式的高可用，因此在节点服务切换过程中 Cinder-volume 服务可能会出现短暂的中断。在本节中，NFS 服务器由宿主机 controller1 实现，三个控制节点分别为 controller1-vm、controller2-vm 和 controller3-vm，Cinder 服务的高可用部署可参考以下实现过程。

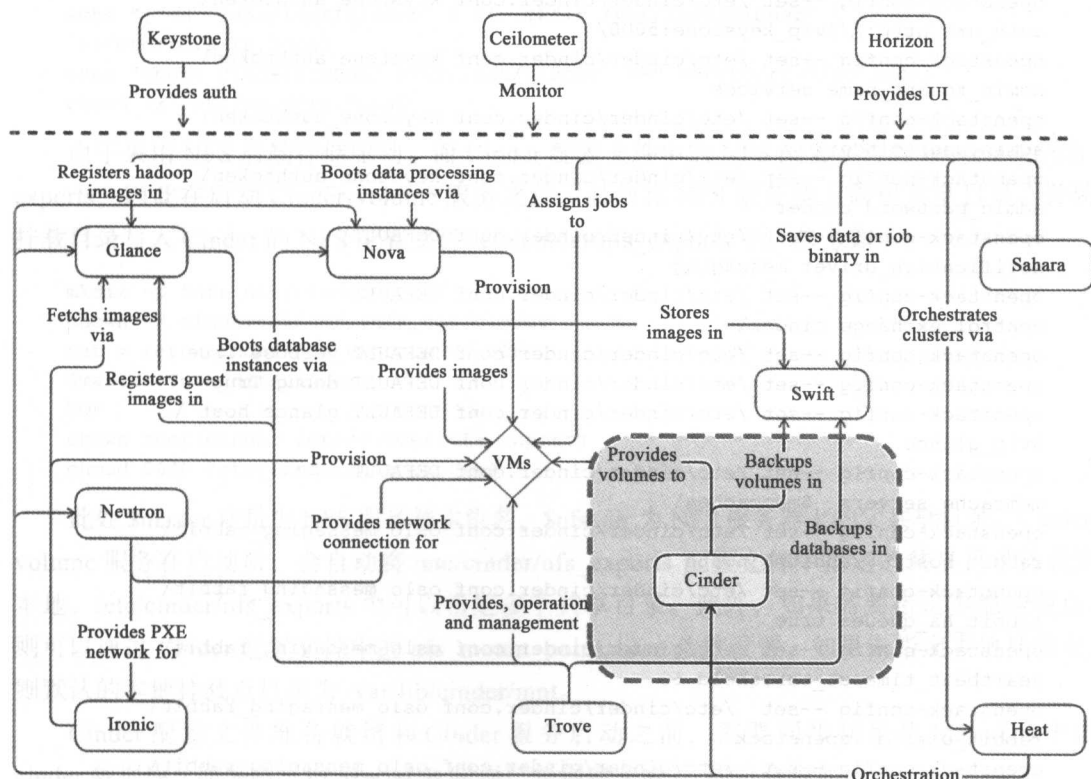


图 12-3 Cinder 在 OpenStack 概念架构中的位置

在三个控制节点上分别安装 Cinder 相关软件包：

① https://bugzilla.redhat.com/show_bug.cgi?id=1193229


```
yum install -y openstack-cinder openstack-utils python-memcached \
python-oslo-db python-cinderclient python-oslo-log MySQL-python \
python-keystonemiddleware python-openstackclient
```

安装完成后，对 Cinder 进行配置，配置文件为 /etc/cinder/cinder.conf，Cinder 的配置文件的多种方法，具体取决于采用什么后端驱动以及是否采用多存储后端，更为详细的 Cinder 配置讲解请参考本书第 10 章的相关内容，这里配置 Cinder-volume 后端存储驱动为 NFS，且采用单一存储后端，参考配置如下：

```
openstack-config --set /etc/cinder/cinder.conf database connection \
mysql://cinder:cinder@$vip_db/cinder
openstack-config --set /etc/cinder/cinder.conf database max_retries -1
openstack-config --set /etc/cinder/cinder.conf DEFAULT auth_strategy \
keystone
openstack-config --set /etc/cinder/cinder.conf keystone_authtoken \
identity_uri http://$vip_keystone:35357/
openstack-config --set /etc/cinder/cinder.conf keystone_authtoken \
auth_uri http://$vip_keystone:5000/
openstack-config --set /etc/cinder/cinder.conf keystone_authtoken \
admin_tenant_name services
openstack-config --set /etc/cinder/cinder.conf keystone_authtoken \
admin_user cinder
openstack-config --set /etc/cinder/cinder.conf keystone_authtoken \
admin_password cinder
openstack-config --set /etc/cinder/cinder.conf DEFAULT \
notification_driver messaging
openstack-config --set /etc/cinder/cinder.conf DEFAULT \
control_exchange cinder \
openstack-config --set /etc/cinder/cinder.conf DEFAULT verbose True
openstack-config --set /etc/cinder/cinder.conf DEFAULT debug True
openstack-config --set /etc/cinder/cinder.conf DEFAULT glance_host \
$vip_glance
openstack-config --set /etc/cinder/cinder.conf DEFAULT \
memcache_servers $memcaches \
openstack-config --set /etc/cinder/cinder.conf oslo_messaging_rabbit \
rabbit_hosts $rabbitmq_hosts
openstack-config --set /etc/cinder/cinder.conf oslo_messaging_rabbit \
rabbit_ha_queues true
openstack-config --set /etc/cinder/cinder.conf oslo_messaging_rabbit \
heartbeat_timeout_threshold 60
openstack-config --set /etc/cinder/cinder.conf oslo_messaging_rabbit \
rabbit_userid openstack
openstack-config --set /etc/cinder/cinder.conf oslo_messaging_rabbit \
rabbit_password openstack
openstack-config --set /etc/cinder/cinder.conf oslo_concurrency \
lock_path /var/lock/cinder
openstack-config --set /etc/cinder/cinder.conf DEFAULT host \
openstack-cinder
openstack-config --set /etc/cinder/cinder.conf DEFAULT \
osapi_volume_listen $public_bind_host_ip
```

```

openstack-config --set /etc/cinder/cinder.conf DEFAULT \
nfs_shares_config /etc/cinder/nfs_exports
openstack-config --set /etc/cinder/cinder.conf DEFAULT \
nfs_sparsed_volumes true
openstack-config --set /etc/cinder/cinder.conf DEFAULT \
nfs_mount_options v3
openstack-config --set /etc/cinder/cinder.conf DEFAULT volume_driver\
cinder.volume.drivers.nfs.NfsDriver

```

在实际部署应用中，Cinder-volume 节点在重启后，/var/lock/cinder 目录通常不会自动创建，因此 Cinder-volume 经常会报错且不能正常提供 Volume 服务，为了解决这个问题，可以让系统开机自动创建 /var/lock/cinder 目录并设置相应的权限，参考如下：

```

[ -d /var/lock/cinder ] || mkdir /var/lock/cinder && chown \
cinder:cinder /var/lock/cinder -R
sed -i '/\var\lock\cinder\d' /etc/rc.d/rc.local
echo " " >>/etc/rc.d/rc.local
echo "[ -d /var/lock/cinder ] || mkdir /var/lock/cinder" \
>>/etc/rc.d/rc.local
echo "chown cinder:cinder /var/lock/cinder -R" >>/etc/rc.d/rc.local
chmod +x /etc/rc.d/rc.local

```

由于采用 NFS 后端存储驱动，而 Cinder 配置文件中的 NFS 配置文件为 /etc/cinder/nfs_exports，因此在启动 Cinder-volume 服务之前，必须在 NFS 服务器上创建共享目录，并将挂载目录写入 Cinder 的 NFS 配置文件 /etc/cinder/nfs_exports 中，参考如下：

```

mkdir -p $nfs_dir/cinder
chown -R cinder:cinder $nfs_dir/cinder
cat > /etc/cinder/nfs_exports << EOF
$master:$nfs_dir/cinder
EOF
chown root:cinder /etc/cinder/nfs_exports
chmod 0640 /etc/cinder/nfs_exports

```

此处 \$master 变量为 NFS 服务器主机名，\$nfs_dir 为 NFS 服务器上的父共享目录。Cinder-volume 服务在启动后，会自动将 /etc/cinder/nfs_exports 配置文件中指定的共享目录挂载到本地，/etc/cinder/nfs_exports 中可以指定多个共享目录。此外，如果需要指定本地挂载点，则可以通过 Cinder 配置文件中的 nfs_mount_point_base 参数设置，如果未指定本地挂载点，则默认的本地挂载点目录为 /var/lib/cinder/mnt。

Cinder 配置文件准备就绪和 Cinder 服务启动之前，需要同步初始化 MariaDB 中的 cinder 数据库（部署高可用 MariaDB 时候已经创建完成），同步命令如下：

```
su -s /bin/sh -c "cinder-manage db sync" cinder
```

Cinder 数据库同步完成后，便可在 Pacemaker 集群中创建 Cinder-api、Cinder-scheduler 和 Cinder-volume 资源，其中 Cinder-api 和 Cinder-scheduler 采用 A/A 高可用模式，而 Cinder-volume 采用 A/P 高可用模式，Pacemaker 集群中的 Cinder 资源按 Keystone→Cinder-

api→Cinder-scheduler→Cinder-volume 顺序启动, 同时 Cinder-api、Cinder-scheduler 和 Cinder-volume 资源需要通过 Pacemaker 的 Colocation 约束绑定到同一个节点上运行, 具体的 Pacemaker 集群资源创建过程参考如下:

```
pcs resource create cinder-api systemd:openstack-cinder-api --clone\
interleave=true
pcs resource create cinder-scheduler systemd:openstack-cinder-\
scheduler --clone interleave=true
//创建约束
pcs resource create cinder-volume systemd:openstack-cinder-volume
pcs constraint order start cinder-api-clone then cinder-scheduler-clone
pcs constraint colocation add cinder-scheduler-clone with \
cinder-api-clone
pcs constraint order start cinder-scheduler-clone then cinder-volume
pcs constraint colocation add cinder-volume with cinder-scheduler-clone
pcs constraint order start keystone-clone then cinder-api-clone
```

Pacemaker 集群中创建完成 Cinder-api、Cinder-scheduler 和 Cinder-volume 资源后, Pacemaker 将会按照 Order 和 Colocation 约束启动与 Cinder 相关的服务。通过 Pacemaker 的 pcs resource 命令可以看到 Pacemaker 集群中的资源运行状态, 如下:

```
[root@controller1-vm ~]# pcs resource
.....
Clone Set: cinder-api-clone [cinder-api]
    Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: cinder-scheduler-clone [cinder-scheduler]
    Started: [ controller1-vm controller2-vm controller3-vm ]
cinder-volume (systemd:openstack-cinder-volume): Started
controller1-vm
```

可以看到, Cinder-api 和 Cinder-scheduler 同时运行在三个控制节点上, 而 Cinder-volume 仅运行在 controller1-vm 控制节点上。上述 Pacemaker 集群中的资源输出结果表明, Cinder 服务已经正常运行, 通过创建 Volume 可以验证 Cinder 服务是否准备就绪, 创建一块 2GB 大小, 名称为 volume1 的块设备, 命令如下:

```
[root@controller1-vm ~(keystone_admin)]$ cinder create --display-name\
volume1 2
[root@controller1-vm ~(keystone_admin)]$ cinder list
```

ID	Status	Display Name	Size	Volume Type	Bootable	Attached to
...6abd5	available	volume1	2	-	false	

因为 Cinder-volume 仅运行在 controller1-vm 控制节点上, 所以与 Cinder 相关的 NFS 共享目录应该已经挂载到 controller1-vm 节点的本地目录上, 通过 df 命令可以查看三个控制节点的目录挂载情况, 如下:

```
[root@controller1-vm ~]# df -h
Filesystem                Size  Used Avail Use% Mounted on
.....
controller1:/data          36G   26G   11G   72% /data
controller1:/data/glance   36G   26G   11G   72% /var/lib/glance
controller1:/data/cinder   36G   26G   11G   72% /var/lib/cinder/mnt/9af0700dd0c3
bf3dc6174bad82dcb7a0
[root@controller2-vm ~]# df -h
Filesystem                Size  Used Avail Use% Mounted on
.....
controller1:/data          36G   30G   6.3G   83% /data
controller1:/data/glance   36G   30G   6.3G   83% /var/lib/glance
[root@controller3-vm ~]# df -h
Filesystem                Size  Used Avail Use% Mounted on
.....
controller1:/data          36G   30G   6.3G   83% /data
controller1:/data/glance   36G   30G   6.3G   83% /var/lib/glance
```

可以看到，NFS 服务器上的共享目录 /data/cinder 仅被挂载到 controller1-vm 节点的本地目录，由于未在配置文件中指定本地挂载点，因此默认挂载到本地 /var/lib/cinder/mnt 目录下。同时，controller2-vm 和 controller3-vm 控制节点因为没有运行 Cinder-volume 服务，所以未挂载 NFS 服务器的 /data/cinder 目录。作为对比，由于 Glance-fs 资源同时运行在三个控制节点，可以看到 NFS 服务上的 /data/glance 共享目录被同时挂载到三个控制节点的本地目录 /var/lib/glance 上。至此，Cinder 块存储服务已通过 Pacemaker 集群实现了服务高可用，后续将配置 OpenStack 集群的高可用网络服务。本节介绍的 Cinder 高可用部署源代码可参考笔者位于 Github 上的开源项目（<https://github.com/ynwssjx/Openstack-HA-Deployment>），具体的部署脚本为 9_create_cinder_resource_on_pacemaker.sh，执行该脚本后，将会在 Pacemaker 集群中创建高可用的 Cinder 块存储服务。

12.1.4 Neutron 网络服务高可用部署

Neutron 网络服务是 OpenStack 开源云计算的核心项目之一，在 OpenStack 云计算环境中，Neutron 项目负责管理虚拟网络基础设施（Virtual Network Infrastructure, VNI）全部网络功能的实现，同时还负责管理与物理网络基础设施（Physical Network Infrastructure, PNI）接入层相关的各种网络功能。通过 Neutron 网络服务，租户可以在 OpenStack 环境中创建各种高级虚拟网络拓扑，如防火墙服务（Firewall）、负载均衡服务（Load balancer）和虚拟私有网络服务（VPN）。

Neutron 将网络、子网和路由器等网络单元进行抽象，在 Neutron 网络中，每个被虚拟化抽象出的对象都有与其物理对象对应的功能，如抽象出的虚拟网络对象可以包含子网，而路由器对象可以转发不同子网与网络之间的数据流。对任意 Neutron 网络设置而言，至少存在一个虚拟外部网络，与其他虚拟网络不同，Neutron 中的外部网络不仅是一个被定义的虚拟网络，还代表了外部物理网络中的一个部分，因此 Neutron 外部网络中的 IP 地址可

以被外部物理网络中的任何客户端访问。除了虚拟外部网络，任何 Neutron 网络设置中至少还存在一个或多个内部网络，内部网络通常属于软件定义网络（SDN），并且主要用于虚拟机的接入。而只有接入相同内部网络的虚拟机之间，或者接入的不同内部网络之间存在路由的情况下虚拟机之间才能相互访问。如果外部物理网络要访问内网中的虚拟机，则网络之间的虚拟路由器是必须的，每个路由器都有一个接入外部网络的网关和一个或多个与内部网络相连的接口。与物理路由器类似，位于当前子网上的虚拟机可以访问连接到同一路由器上的虚拟机，而虚拟机可以通过路由器的网关访问外部网络。此外，用户可以将外部网络上的 IP 地址分配给内网端口（一旦有任何网络对象连接到子网上，则该连接便称为端口），因此用户可以将外网 IP 地址关联到内网虚拟机上，从而外网客户端可以直接与虚拟机通信。

Neutron 网络还支持安全组（Security Group），安全组可以让用户将防火墙规则编辑到一个组中，而一个虚拟机可以属于一个或多个安全组。Neutron 网络将根据安全组中的规则来对虚拟机端口、端口范围或网络类型进行阻止或取消阻止操作。Neutron 支持不同的网络插件，而每个网络插件都有自己的概念，尽管网络插件的概念对于 VNI 和 OpenStack 的操作并非十分重要，但是理解网络插件的概念对于 Neutron 的网络设置十分重要，在 Neutron 中，所有网络安装配置都需要使用一个核心插件（如 ML2）和一个安全组插件。插件架构设计是 Neutron 网络能够适应不同网络设备和软件环境以及满足不同网络拓扑需求的基础，插件设计使得 OpenStack 可以部署出灵活多变的网络架构。

Neutron 主要由 Neutron-server 和各种网络插件（Plugin）及代理（Agent）构成，Neutron-server 主要负责接受客户端请求并将其转发给后端网络插件进行处理，而插件则负责具体的网络功能实现和网络对象的创建，如创建网络、子网和提供 IP 地址等。插件和代理的功能依据特定云环境所使用的不同厂商插件和网络技术而有所区别，OpenStack 网络自带有很多厂商或开源网络插件，如 OpenvSwitch、Linux bridging 以及 Cisco 的虚拟和物理交换机插件、NEC Openflow 插件和 VMware NSX 插件等，用户可以根据自己的云环境选择使用不同的网络插件，从而部署出适合自身网络环境的 Neutron 网络，对于很多开源用户而言，OpenvSwitch 和 Linux bridging 是使用最多也是较为成熟的插件。网络代理（Agent）主要包括 L3 Agent、DHCP Agent 和相应的插件代理，其中，L3 Agent 是影响 Neutron 网络高可用设计的关键，也是 Neutron 网络服务高可用部署的难点和重点。Neutron 在 OpenStack 开源云计算概念架构设计中的位置及与其他组件的交互如图 12-4 所示。

根据本书第 9 章的介绍，Neutron 网络服务的高可用可以分为三种实现方案，即基于虚拟路由冗余协议（VRRP）的 L3 HA 方案、基于分布式虚拟路由（DVR）的方案以及结合 L3 HA 和 DVR 实现的网络高可用方案。本节将采用 L3 HA 高可用模式部署 Neutron 网络，将 OpenStack 三个控制节点作为 Neutron 高可用网络节点，同时将 Neutron-server 服务以 A/A 模式运行在三个控制节点（网络节点）上，并在每个网络节点上取用 L3_HA 功能，使得租户创建的每个 L3 路由均分布在三个网络节点上，并且每个租户网络均有三个 DHCP 服务与

之对应，Neutron 网络服务高可用部署过程可参考以下实现。

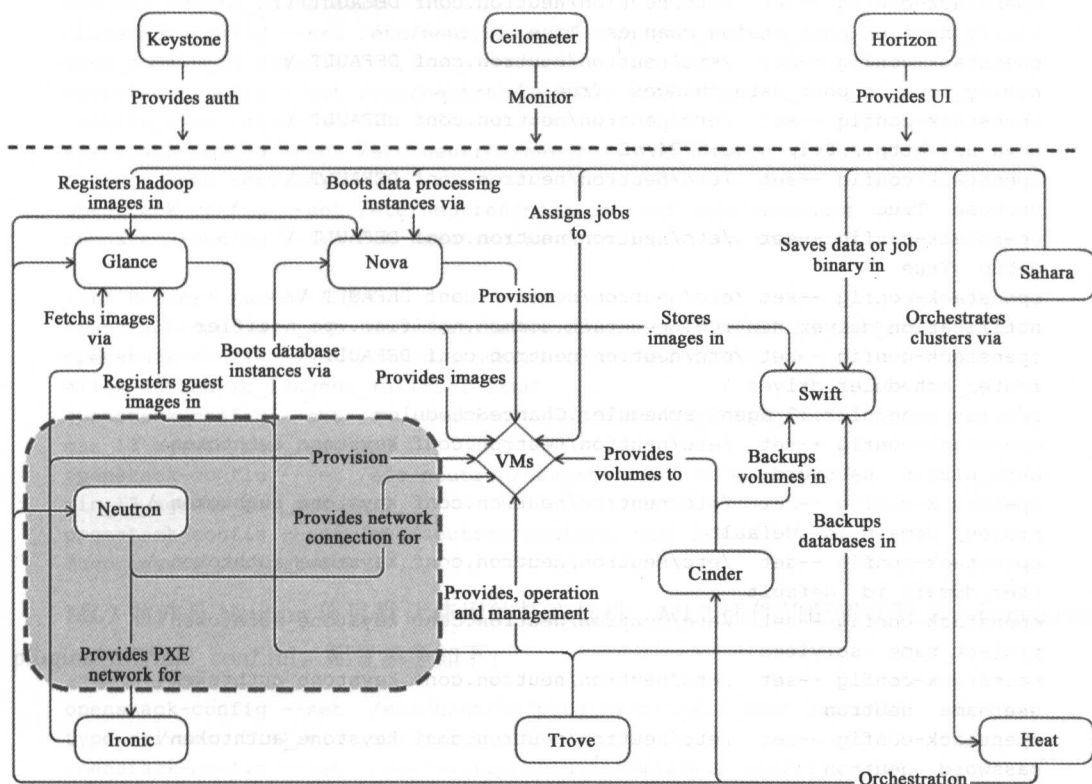


图 12-4 Neutron 在 OpenStack 概念架构设计中的位置

在三个控制节点上安装部署 Neutron 相关软件包：

```

yum install -y openstack-neutron openstack-neutron-openvswitch \
openstack-neutron-ml2 openvswitch python-neutronclient \
python-openstackclient which

```

安装完成之后，对 Neutron-server 进行配置，配置文件为 /etc/neutron/neutron.conf，配置参考如下：

```

openstack-config --set /etc/neutron/neutron.conf DEFAULT \
bind_host $public_bind_host_ip
openstack-config --set /etc/neutron/neutron.conf DEFAULT \
rpc_backend rabbit
openstack-config --set /etc/neutron/neutron.conf DEFAULT \
auth_strategy keystone
openstack-config --set /etc/neutron/neutron.conf DEFAULT \
core_plugin ml2
openstack-config --set /etc/neutron/neutron.conf DEFAULT \
service_plugins router
openstack-config --set /etc/neutron/neutron.conf DEFAULT \

```



```

allow_overlapping_ips True
openstack-config --set /etc/neutron/neutron.conf DEFAULT \
notify_nova_on_port_status_changes True
openstack-config --set /etc/neutron/neutron.conf DEFAULT \
notify_nova_on_port_data_changes True
openstack-config --set /etc/neutron/neutron.conf DEFAULT \
nova_url http://$vip_nova:8774/v2
openstack-config --set /etc/neutron/neutron.conf DEFAULT \
verbose True
openstack-config --set /etc/neutron/neutron.conf DEFAULT \
debug True
openstack-config --set /etc/neutron/neutron.conf DEFAULT \
notification_driver neutron.openstack.common.notifier.rpc_notifier
openstack-config --set /etc/neutron/neutron.conf DEFAULT \
router_scheduler_driver \
neutron.scheduler.l3_agent_scheduler.ChanceScheduler
openstack-config --set /etc/neutron/neutron.conf keystone_authtoken \
auth_plugin password
openstack-config --set /etc/neutron/neutron.conf keystone_authtoken \
project_domain_id default
openstack-config --set /etc/neutron/neutron.conf keystone_authtoken \
user_domain_id default
openstack-config --set /etc/neutron/neutron.conf keystone_authtoken \
project_name services
openstack-config --set /etc/neutron/neutron.conf keystone_authtoken \
username neutron
openstack-config --set /etc/neutron/neutron.conf keystone_authtoken \
password neutron
openstack-config --set /etc/neutron/neutron.conf keystone_authtoken \
auth_url http://$vip_keystone:35357
openstack-config --set /etc/neutron/neutron.conf keystone_authtoken \
auth_uri http://$vip_keystone:5000
openstack-config --set /etc/neutron/neutron.conf database connection \
mysql://neutron:neutron@$vip_db:3306/neutron
openstack-config --set /etc/neutron/neutron.conf database \
max_retries -1
openstack-config --set /etc/neutron/neutron.conf \
oslo_messaging_rabbit rabbit_hosts $rabbitmq_hosts
openstack-config --set /etc/neutron/neutron.conf \
oslo_messaging_rabbit rabbit_ha_queues true
openstack-config --set /etc/neutron/neutron.conf \
oslo_messaging_rabbit heartbeat_timeout_threshold 60
openstack-config --set /etc/neutron/neutron.conf \
oslo_messaging_rabbit rabbit_userid openstack
openstack-config --set /etc/neutron/neutron.conf \
oslo_messaging_rabbit rabbit_password openstack

openstack-config --set /etc/neutron/neutron.conf nova auth_url \
http://$vip_keystone:35357/
openstack-config --set /etc/neutron/neutron.conf nova auth_plugin \
password

```

```

openstack-config --set /etc/neutron/neutron.conf nova \
project_domain_id default
openstack-config --set /etc/neutron/neutron.conf nova \
user_domain_id default
openstack-config --set /etc/neutron/neutron.conf nova \
region_name regionOne
openstack-config --set /etc/neutron/neutron.conf nova \
project_name services
openstack-config --set /etc/neutron/neutron.conf nova username nova
openstack-config --set /etc/neutron/neutron.conf nova password nova

```

//L3 Router与DHCP高可用设置

```

openstack-config --set /etc/neutron/neutron.conf DEFAULT l3_ha True
openstack-config --set /etc/neutron/neutron.conf DEFAULT \
allow_automatic_l3agent_failover True
openstack-config --set /etc/neutron/neutron.conf DEFAULT \
max_l3_agents_per_router 3
openstack-config --set /etc/neutron/neutron.conf DEFAULT \
min_l3_agents_per_router 2
openstack-config --set /etc/neutron/neutron.conf DEFAULT \
dhcp_agents_per_network 3

```

ML2 插件是 Neutron 项目默认使用的核心插件，ML2 插件的配置文件为 `/etc/neutron/plugins/ml2/ml2_conf.ini`，配置参考如下：

```

openstack-config --set /etc/neutron/plugins/ml2/ml2_conf.ini ml2 \
type_drivers flat,vlan,gre,vxlan
openstack-config --set /etc/neutron/plugins/ml2/ml2_conf.ini ml2 \
tenant_network_types gre
openstack-config --set /etc/neutron/plugins/ml2/ml2_conf.ini ml2 \
mechanism_drivers openvswitch
openstack-config --set /etc/neutron/plugins/ml2/ml2_conf.ini \
ml2_type_gre tunnel_id_ranges 1:1000
openstack-config --set /etc/neutron/plugins/ml2/ml2_conf.ini \
securitygroup enable_security_group True
openstack-config --set /etc/neutron/plugins/ml2/ml2_conf.ini \
securitygroup enable_ipset True
openstack-config --set /etc/neutron/plugins/ml2/ml2_conf.ini \
securitygroup firewall_driver neutron.agent.linux \
iptables_firewall.OVSHybridIptablesFirewallDriver
openstack-config --set /etc/neutron/plugins/ml2/ml2_conf.ini \
ml2_type_flat flat_networks external
openstack-config --set /etc/neutron/plugins/ml2/ml2_conf.ini ovs \
bridge_mappings external:br-ex
openstack-config --set /etc/neutron/plugins/ml2/ml2_conf.ini ovs \
local_ip $public_bind_host_ip
openstack-config --set /etc/neutron/plugins/ml2/ml2_conf.ini agent \
tunnel_types gre
ln -s /etc/neutron/plugins/ml2/ml2_conf.ini /etc/neutron/plugin.ini

```

配置 metadata agent，配置文件为 `/etc/neutron/metadata_agent.ini`，配置参考如下：

```

openstack-config --set /etc/neutron/metadata_agent.ini DEFAULT \
auth_strategy keystone
openstack-config --set /etc/neutron/metadata_agent.ini DEFAULT \
auth_uri http://$vip_keystone:5000
openstack-config --set /etc/neutron/metadata_agent.ini DEFAULT \
auth_url http://$vip_keystone:35357
openstack-config --set /etc/neutron/metadata_agent.ini DEFAULT \
auth_host $vip_keystone
openstack-config --set /etc/neutron/metadata_agent.ini DEFAULT \
auth_region regionOne
openstack-config --set /etc/neutron/metadata_agent.ini DEFAULT \
admin_tenant_name services
openstack-config --set /etc/neutron/metadata_agent.ini DEFAULT \
admin_user neutron
openstack-config --set /etc/neutron/metadata_agent.ini DEFAULT \
admin_password neutron
openstack-config --set /etc/neutron/metadata_agent.ini DEFAULT \
project_domain_id default
openstack-config --set /etc/neutron/metadata_agent.ini DEFAULT \
user_domain_id default
openstack-config --set /etc/neutron/metadata_agent.ini DEFAULT \
nova_metadata_ip $vip_nova
openstack-config --set /etc/neutron/metadata_agent.ini DEFAULT \
nova_metadata_port 8775
openstack-config --set /etc/neutron/metadata_agent.ini DEFAULT \
metadata_proxy_shared_secret $metadata_shared_secret
openstack-config --set /etc/neutron/metadata_agent.ini DEFAULT \
metadata_workers 4
openstack-config --set /etc/neutron/metadata_agent.ini DEFAULT \
metadata_backlog 2048
openstack-config --set /etc/neutron/metadata_agent.ini DEFAULT \
verbose True
openstack-config --set /etc/neutron/metadata_agent.ini DEFAULT \
debug True

```

配置 DHCP Agent, 配置文件为 /etc/neutron/dhcp_agent.ini, 配置参考如下:

```

openstack-config --set /etc/neutron/dhcp_agent.ini DEFAULT \
interface_driver neutron.agent.linux.interface.OVSInterfaceDriver
openstack-config --set /etc/neutron/dhcp_agent.ini DEFAULT \
dhcp_delete_namespaces False
openstack-config --set /etc/neutron/dhcp_agent.ini DEFAULT \
dhcp_driver neutron.agent.linux.dhcp.Dnsmasq
openstack-config --set /etc/neutron/dhcp_agent.ini DEFAULT \
dnsmasq_config_file /etc/neutron/dnsmasq-neutron.conf
echo "dhcp-option-force=26,1454" >/etc/neutron/dnsmasq-neutron.conf
pkill dnsmasq

```

配置 L3 Agent, 配置文件为 /etc/neutron/l3_agent.ini, 配置参考如下:

```

openstack-config --set /etc/neutron/l3_agent.ini DEFAULT \
interface_driver neutron.agent.linux.interface.OVSInterfaceDriver

```

```

openstack-config --set /etc/neutron/l3_agent.ini DEFAULT \
handle_internal_only_routers True
openstack-config --set /etc/neutron/l3_agent.ini DEFAULT \
send_arp_for_ha 3
openstack-config --set /etc/neutron/l3_agent.ini DEFAULT \
router_delete_namespaces False
openstack-config --set /etc/neutron/l3_agent.ini DEFAULT \
external_network_bridge br-ex
openstack-config --set /etc/neutron/l3_agent.ini DEFAULT verbose True
openstack-config --set /etc/neutron/l3_agent.ini DEFAULT debug True
cp /usr/lib/systemd/system/neutron-openvswitch-agent.service\
/usr/lib/systemd/system/neutron-openvswitch-agent.service.orig
sed -i 's,plugins/openvswitch/ovs_neutron_plugin.ini,plugin.ini,g'\
/usr/lib/systemd/system/neutron-openvswitch-agent.service

```

启动 OpenVswitch，添加外部网桥，绑定外网物理端口，参考如下：

```

systemctl enable openvswitch.service
systemctl start openvswitch.service
ovs-vsctl add-br br-int
ovs-vsctl add-br br-ex
//变量$ext_br_nic为控制节点（网络节点）指定的外网物理网卡名称
ovs-vsctl add-port br-ex $ext_br_nic
ethtool -K $ext_br_nic gro off

```

在 MariaDB 中设置用户 neutron 对数据库 neutron（MariaDB 高可用部署阶段已经创建）的访问权限，并对 neutron 数据库进行同步初始化，参考如下：

```

//neutron数据库权限设置
mysql -uroot -proot -e "GRANT ALL PRIVILEGES ON neutron.* TO \
'neutron'@'localhost' IDENTIFIED BY 'neutron';"
mysql -uroot -proot -e "GRANT ALL PRIVILEGES ON neutron.* TO \
'neutron'@'%' IDENTIFIED BY 'neutron';"
//同步初始化neutron数据库
su -s /bin/sh -c "neutron-db-manage --config-file \
/etc/neutron/neutron.conf --config-file\
/etc/neutron/plugins/ml2/ml2_conf.ini upgrade head" neutron

```

Neutron 的配置文件和数据库均准备就绪后，便可在 Pacemaker 集群中创建 Neutron 相关的服务资源。需要指出的是，本章中的 Openstack 完全基于 RDO 发行版 RPM 包进行安装，在 RDO 发行版本中，安装完 Neutron 相关软件包后，在系统 OCF 脚本目录中会增加 Neutron 相关的 OCF 脚本，如下：

```

[root@controller1-vm ~]# cd /usr/lib/ocf/resource.d/neutron
[root@controller1-vm neutron]# ls -l
total 24
-rwxr-xr-x 1 root root 4640 Apr 25 2015 NetnsCleanup
-rwxr-xr-x 1 root root 5831 Apr 26 2015 NeutronScale
-rwxr-xr-x 1 root root 4606 Apr 25 2015 OVSCleanup

```

可以看到，系统 OCF 类资源中增加了 Provider 为 Neutron，Type 分别为 NetnsCleanup、

NeutronScale 和 OVSCleanup 的资源代理 (Resource Agent, RA)。其中, NetnsCleanup 和 OVSCleanup 分别用于在 Neutron 的 Agent 服务启停过程中清除节点上的网络命名空间和 OVS 信息。NeutronScale 是由 RedHat 开发, 并用于动态命名 Neutron 代理服务所使用主机名的 RA, 即 NeutronScale 资源代理会动态更改各个网络节点中与 Neutron 相关的 .conf 和 .ini 配置文件, 并在其中 DEAFULT 配置段设置虚拟 host 参数 (将 host 命名为 neutron-n-x, x=1, 2, 3, ...), 即主机虚拟 ID, 从而保证 Neutron Agent 故障切换时无须重新在目标主机上构建 Neutron Agent 服务, 而是直接使用目标主机上已存在的 Agent 提供服务。在使用 NeutronScale 资源代理的 Pacemaker 集群中, Neutron 资源正常运行后, 使用命令 neutron agent-list 看到的 host 字段将不是真实的网络节点主机名, 而是 NeutronScale 所设置的虚拟主机名。在本章的三控制节点 Pacemaker 集群中使用 NeutronScale 资源代理部署 Neutron 高可用网络后, Neutron 各个 Agent 运行情况如下:

```
root@controller1-vm ~(keystone_admin)]$ neutron agent-list
+-----+-----+-----+-----+-----+
| id | agent_type | host | alive | binary |
+-----+-----+-----+-----+-----+
|...a0|Metadata agent | neutron-n-0 | :- ) | neutron-metadata-agent |
|...4d|DHCP agent | neutron-n-0 | :- ) | neutron-dhcp-agent |
|...8a|L3 agent | neutron-n-1 | :- ) | neutron-l3-agent |
|...22|DHCP agent | neutron-n-2 | :- ) | neutron-dhcp-agent |
|...1a|Metadata agent | neutron-n-2 | :- ) | neutron-metadata-agent |
|...a9|L3 agent | neutron-n-2 | :- ) | neutron-l3-agent |
|...1a|Open vSwitch agent| neutron-n-0 | :- ) | neutron-openvswitch-agent|
|...9d|L3 agent | neutron-n-0 | :- ) | neutron-l3-agent |
|...21|Open vSwitch agent| neutron-n-2 | :- ) | neutron-openvswitch-agent|
|...45|DHCP agent | neutron-n-1 | :- ) | neutron-dhcp-agent |
|...16|Open vSwitch agent| neutron-n-1 | :- ) | neutron-openvswitch-agent|
|...29|Metadata agent | neutron-n-1 | :- ) | neutron-metadata-agent |
+-----+-----+-----+-----+-----+
```

NeutronScale 资源在 Pacemaker 集群中的运行情况如下:

```
root@controller1-vm ~(keystone_admin)]$ pcs status |grep neutron -A 2
.....
Clone Set: neutron-scale-clone [neutron-scale]
    neutron-scale:0 (ocf::neutron:NeutronScale): Started controller2-vm
    neutron-scale:1 (ocf::neutron:NeutronScale): Started controller3-vm
    neutron-scale:2 (ocf::neutron:NeutronScale): Started controller1-vm
.....
```

可以看到, NeutronScale 对于控制节点虚拟主机的命名是随机的, 即虚拟主机名与真实主机名之间并不存在固定的映射关系, 而这种动态的映射关系在 Neutron 服务重启之后经常与数据库中静态的映射表之间产生冲突[⊖], 这种冲突关系如表 12-1 所示。

⊖ <https://bugs.launchpad.net/neutron/+bug/1464178>

表 12-1 NeutronScale 资源造成的真实主机映射与数据库表映射冲突关系

实际映射关系		数据库表中映射关系	
neutron-n-0	controller2-vm	neutron-n-0	controller2-vm
neutron-n-1	controller3-vm	neutron-n-1	controller1-vm
neutron-n-2	controller1-vm	neutron-n-2	controller3-vm

对于 GRE 类型网络，neutron 数据库中保存主机映射关系的表是 ml2_gre_endpoints，而对于 VxLAN 网络，则是 ml2_vxlan_endpionts。通常解决这种冲突关系的方法就是手动清除上述两个表中的全部记录，然后再重新启动 Neutron 服务。而 NeutronScale 资源引起的另外一个弊端，是在 NeutronScale 启动后，可能存在的竞争关系使得 Neutron 服务进入一种糟糕的状态^①，此外，NeutronScale 还会造成持有 Active 状态 Router 的主机动态改变，致使管理员无法定位当前活动的网络节点。介于上述原因，在使用 RDO 部署 OpenStack 高可用集群时，强烈建议部署 Kilo 以上版本，因为 Kilo 以上 OpenStack 版本解决了 Neutron 高可用网络环境中引入 NeutronScale 的必要性，即 Kilo 以上版本在 Pacemaker 集群中部署高可用网络时完全无须再使用 NeutronScale 资源代理。鉴于此，本节在 Pacemaker 集群中部署 Neutron 相关资源时，将不会再使用 NeutronScale，具体的 Pacemaker 资源创建、Order 约束和 Colocation 约束设置过程参考如下：

//创建Neutron资源

```
pcs resource create neutron-server-api systemd:neutron-server op start\
timeout=180 --clone interleave=true
pcs resource create neutron-ovs-cleanup ocf:neutron:OVSCleanup\
--clone interleave=true
pcs resource create neutron-netns-cleanup ocf:neutron:NetnsCleanup\
--clone interleave=true
pcs resource create neutron-openvswitch-agent \
systemd:neutron-openvswitch-agent --clone interleave=true
pcs resource create neutron-dhcp-agent systemd:neutron-dhcp-agent \
--clone interleave=true
pcs resource create neutron-l3-agent systemd:neutron-l3-agent\
--clone interleave=true
pcs resource create neutron-metadata-agent \
systemd:neutron-metadata-agent --clone interleave=true
//设置Order约束
pcs constraint order start keystone-clone then neutron-server-api-clone
pcs constraint order start neutron-server-api-clone then \
neutron-ovs-cleanup-clone
pcs constraint order start neutron-ovs-cleanup-clone then \
neutron-netns-cleanup-clone
pcs constraint order start neutron-netns-cleanup-clone then \
neutron-openvswitch-agent-clone
pcs constraint order start neutron-openvswitch-agent-clone then \
```

① https://bugzilla.redhat.com/show_bug.cgi?id=1238117


```

neutron-dhcp-agent-clone
pcs constraint order start neutron-dhcp-agent-clone then \
neutron-l3-agent-clone
pcs constraint order start neutron-l3-agent-clone then \
neutron-metadata-agent-clone
//设置Colocation约束
pcs constraint colocation add neutron-netns-cleanup-clone with \
neutron-ovs-cleanup-clone
pcs constraint colocation add neutron-openvswitch-agent-clone with \
neutron-netns-cleanup-clone
pcs constraint colocation add neutron-dhcp-agent-clone with \
neutron-openvswitch-agent-clone
pcs constraint colocation add neutron-l3-agent-clone with \
neutron-dhcp-agent-clone
pcs constraint colocation add neutron-metadata-agent-clone with \
neutron-l3-agent-clone

```

Neutron 资源创建完成后, Pacemaker 将根据 Order 和 Colocation 约束依次启动 Neutron-server-api 服务、neutron-ovs-cleanup 服务、neutron-netns-cleanup 服务、neutron-openvswitch-agent 服务、neutron-dhcp-agent 服务、neutron-l3-agent 服务和 neutron-metadata-agent 服务。服务资源启动完成后, 在 Pacemaker 集群中通过 pcs resource 命令即可查看 Neutron 资源的运行情况, 如下:

```

[root@controller1-vm ~]# pcs resource
.....
Clone Set: neutron-server-api-clone [neutron-server-api]
  Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: neutron-ovs-cleanup-clone [neutron-ovs-cleanup]
  Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: neutron-netns-cleanup-clone [neutron-netns-cleanup]
  Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: neutron-openvswitch-agent-clone
[neutron-openvswitch-agent]
  Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: neutron-dhcp-agent-clone [neutron-dhcp-agent]
  Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: neutron-l3-agent-clone [neutron-l3-agent]
  Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: neutron-metadata-agent-clone [neutron-metadata-agent]
  Started: [ controller1-vm controller2-vm controller3-vm ]

```

可以看到, Pacemaker 集群中全部 Neutron 资源均以 A/A 高可用模式运行在三个控制节点上, 为了验证 Neutron 服务是否如 Pacemaker 集群中的资源所示, 可以通过 Neutron 命令行客户端来检查 Neutron Agent 的运行状态, 如下:

```

[root@controller1-vm ~(keystone_admin)]$ neutron agent-list
+-----+-----+-----+-----+-----+
| id | agent_type | host | alive | binary |
+-----+-----+-----+-----+-----+

```

```

|...a78 | DHCP agent | controller1-vm | :-) | neutron-dhcp-agent |
|...96f | Metadata agent | controller2-vm | :-) | neutron-metadata-agent |
|...799 | Open vSwitch agent | controller1-vm | :-) | neutron-openvswitch-agent |
|...c55 | DHCP agent | controller2-vm | :-) | neutron-dhcp-agent |
|...38c | Open vSwitch agent | controller2-vm | :-) | neutron-openvswitch-agent |
|...cf7 | L3 agent | controller1-vm | :-) | neutron-l3-agent |
|...840 | Open vSwitch agent | controller3-vm | :-) | neutron-openvswitch-agent |
|...1f9 | Metadata agent | controller3-vm | :-) | neutron-metadata-agent |
|...979 | L3 agent | controller2-vm | :-) | neutron-l3-agent |
|...d71 | DHCP agent | controller3-vm | :-) | neutron-dhcp-agent |
|...75d | L3 agent | controller3-vm | :-) | neutron-l3-agent |
|...cf5 | Metadata agent | controller1-vm | :-) | neutron-metadata-agent |
+-----+-----+-----+-----+-----+

```

可以看到, Neutron 的 L3 Agent、DHCP Agent、Metadata Agent 和 Open vSwitch Agent 均正常运行在三个控制节点上。为了验证 Neutron 网络服务已经准备就绪, 可以在 Neutron 中创建网络、子网和路由等网络对象单元, 以检查 Neutron 网络功能的正常性。如下代码段将在 Neutron 网络中创建一个名为 ext-net 的外网及其子网 ext-subnet, 同时创建一个名为 admin-net 的内网及其子网 admin-subnet, 此外还创建一个名为 admin-router 的路由并为其设置网关和内网接口, 代码参考如下:

```

source /root/adminrc
neutron net-create ext-net --router:external \
--provider:physical_network external --provider:network_type flat
neutron subnet-create ext-net 192.168.115.0/24 --name ext-subnet\
--allocation-pool start=192.168.115.200,end=192.168.115.250
--disable-dhcp --gateway 192.168.115.254
neutron net-create admin-net
neutron subnet-create admin-net 192.128.1.0/24 --name admin-subnet\
--gateway 192.128.1.1
neutron router-create admin-router
neutron router-interface-add admin-router admin-subnet
neutron router-gateway-set admin-router ext-net

```

上述代码运行完成后, Neutron 中应该存在三个网络, 其中两个为租户网络, 另一个为 Neutron 自动创建的 HA 网络, 如下:

```

[root@controller1-vm ~(keystone_admin)]$ neutron net-list
+-----+-----+-----+-----+-----+
| id | name | subnets |
+-----+-----+-----+-----+
|...bad | ext-net | ...549 192.168.115.0/24 |
|...ad3 | HA network tenant ...3d5135 | ...bf7 169.254.192.0/18 |
|...57e | admin-net | ...f8f 192.128.1.0/24 |
+-----+-----+-----+-----+

```

Neutron 网络中的 Router 信息如下, 注意其中的 HA 属性是否为 True:

```

[root@controller1-vm ~(keystone_admin)]$ neutron router-list

```

```

+-----+-----+-----+-----+-----+
| id | name | external_gateway_info | distributed | ha |
+-----+-----+-----+-----+-----+
| ...784 | admin-router | {...} | False | True |
+-----+-----+-----+-----+-----+

```

根据 L3 Agent 和 DHCP Agent 的高可用配置，用户创建的 Router 应该分布在三个网络节点的命名空间中，同时针对租户创建的每个 DHCP 内部网络，网络节点命名空间中应该存在三个 DHCP 服务器，从而保证任意网络节点（控制节点）故障，L3 Agent 和 DHCP Agent 均不受影响。三个控制节点网络命名空间中 L3 Router 和 DHCP Server 的情况如下：

```

[root@controller1-vm ~]# ip netns
qrouter-0d022a43-2d8d-492c-9f33-04d78cce0784
qdhcp-dfb56774-27af-4059-b634-990b20eef57e
[root@controller2-vm ~]# ip netns
qrouter-0d022a43-2d8d-492c-9f33-04d78cce0784
qdhcp-dfb56774-27af-4059-b634-990b20eef57e
[root@controller3-vm ~]# ip netns
qrouter-0d022a43-2d8d-492c-9f33-04d78cce0784
qdhcp-dfb56774-27af-4059-b634-990b20eef57e

```

可以看到，三个控制节点网络命名空间中分别存在三个 ID 完全相同的 qrouter 和 qdhcp 对象，说明 L3 Router 和 DHCP Agent 已经成功以副本形式分布到三个控制节点，从而保证了租户网络功能的高可用性。至此，Neutron 网络功能高可用配置部署已经完成，后续将继续实现计算服务 Nova 的高可用部署。本节介绍的 Neutron 高可用部署源代码可参考笔者位于 Github 上的开源项目（<https://github.com/ynwssjx/Openstack-HA-Deployment>），具体的部署脚本为 10_create_neutron_resource_on_pacemaker.sh，执行该脚本后，将会在 Pacemaker 集群中创建高可用的 Neutron 网络服务。

12.1.5 Nova API 服务高可用部署

Nova 是 OpenStack 所有项目中最为核心和成熟的项目，OpenStack 中很多新增项目是由 Nova 发展而来或是对 Nova 的补充，Nova 负责为用户提供 OpenStack 云平台中最为重要的部分，即计算服务。在 OpenStack 项目中，Nova 与很多服务项目均有交互，Nova 在 OpenStack 概念架构设计中的位置及与其他项目的交互如图 12-5 所示。在实际应用中，当租户发起虚拟机创建请求时，Nova 首先需要与 Glance 服务交互以从中获取虚拟机镜像，同时需要与 Neutron 网络服务交互以使其为 Nova 虚拟机提供通信服务，并且还需与 Cinder/Swift 存储服务交互以使其为 Nova 虚拟机提供持久性存储服务，此外，Nova 与 Keystone 和 Horizon 等其他 OpenStack 服务项目均有不同程度的交互。Nova 是 OpenStack 众多项目的服务中心，失去了 Nova 提供的计算服务，OpenStack 也就失去了开源云计算的意义。

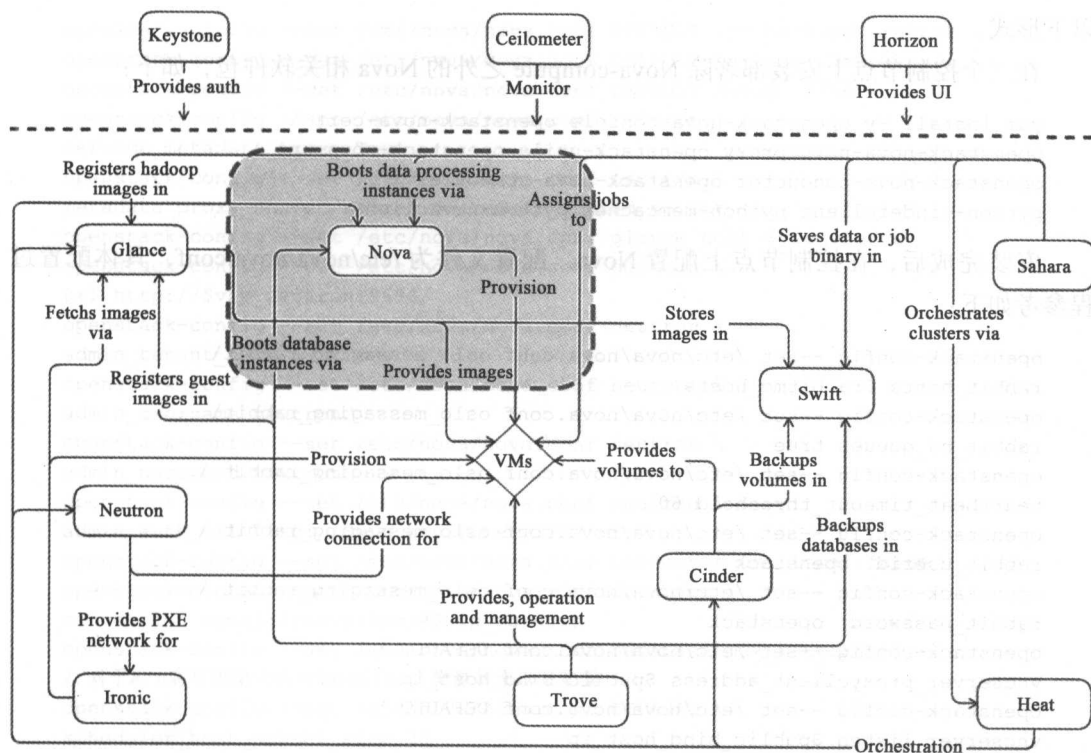


图 12-5 Nova 在 OpenStack 概念架构设计中的位置

计算服务 Nova 由很多子服务组成，尽管 Nova-volume 和 Nova-network 已经独立发展成为 Cinder 和 Neutron 两个核心项目，但是 Nova 依然是目前 OpenStack 中服务组件的构成较为繁杂的项目。在 Nova 的诸多服务项目中，Nova-api、Nova-scheduler、Nova-conductor 和 Nova-compute 是最为关键和重要的服务，除此之外，Nova-consoleauth、Nova-cert、Nova-novncproxy 也是经常使用到的 Nova 服务。在 Nova 的各种服务中，除 Nova-compute 主要用于虚拟机的供给（Provision）之外，其他服务主要负责实现 Nova 客户端的请求响应、调度转发以及 Nova 内部的访问认证、元数据处理和数据库访问隔离等功能，这些服务均为无状态服务，通常部署在控制节点，而 Nova-compute 服务通常部署在独立计算节点上以提供虚拟机实例服务。

在实际的 Nova 部署过程中，尽管 Nova-compute 也可以采用“All in one”的方式部署在控制节点上，但是通常的控制节点已经部署太多 OpenStack 服务而承载了较重的运行负荷，因此不建议控制节点同时充当计算节点。本节暂时仅实现 Nova-api、Nova-scheduler 和 Nova-conductor 等控制节点 API 服务的高可用部署，而 Nova-compute 计算服务将部署在独立计算节点上，其部署过程将在控制节点服务高可用部署完成后再进行。由于 Nova 的 API 服务均为无状态服务，因此可以直接在 Pacemaker 集群中以 Clone 资源形式运行在三个控制节点上，并通过 HAProxy 实现 A/A 高可用。Nova 控制节点服务高可用部署过程可参考

以下形式。

在三个控制节点上安装部署除 Nova-compute 之外的 Nova 相关软件包，如下：

```
yum install -y openstack-nova-console openstack-nova-cert \
openstack-nova-novncproxy openstack-utils openstack-nova-api \
openstack-nova-conductor openstack-nova-scheduler \
python-cinderclient python-memcached python-novaclient
```

安装完成后，在控制节点上配置 Nova，配置文件为 /etc/nova/nova.conf，具体配置过程参考如下：

```
openstack-config --set /etc/nova/nova.conf oslo_messaging_rabbit \
rabbit_hosts $rabbitmq_hosts
openstack-config --set /etc/nova/nova.conf oslo_messaging_rabbit \
rabbit_ha_queues true
openstack-config --set /etc/nova/nova.conf oslo_messaging_rabbit \
heartbeat_timeout_threshold 60
openstack-config --set /etc/nova/nova.conf oslo_messaging_rabbit \
rabbit_userid openstack
openstack-config --set /etc/nova/nova.conf oslo_messaging_rabbit \
rabbit_password openstack
openstack-config --set /etc/nova/nova.conf DEFAULT \
vncserver_proxyclient_address $public_bind_host_ip
openstack-config --set /etc/nova/nova.conf DEFAULT \
vncserver_listen $public_bind_host_ip
openstack-config --set /etc/nova/nova.conf DEFAULT \
vncserver_proxyclient_address $public_bind_host_ip
openstack-config --set /etc/nova/nova.conf DEFAULT \
novncproxy_host $public_bind_host_ip
openstack-config --set /etc/nova/nova.conf DEFAULT \
metadata_listen $public_bind_host_ip
openstack-config --set /etc/nova/nova.conf DEFAULT \
osapi_compute_listen $public_bind_host_ip
openstack-config --set /etc/nova/nova.conf DEFAULT \
novncproxy_base_url http://$vip_nova:6080/vnc_auto.html
openstack-config --set /etc/nova/nova.conf DEFAULT \
auth_strategy keystone
openstack-config --set /etc/nova/nova.conf DEFAULT \
memcached_servers $memcaches
openstack-config --set /etc/nova/nova.conf DEFAULT vnc_enabled True
openstack-config --set /etc/nova/nova.conf DEFAULT \
metadata_host $vip_nova
openstack-config --set /etc/nova/nova.conf DEFAULT \
metadata_listen_port 8775
openstack-config --set /etc/nova/nova.conf DEFAULT \
firewall_driver nova.virt.firewall.NoopFirewallDriver
openstack-config --set /etc/nova/nova.conf DEFAULT \
libvirt_vif_driver nova.virt.libvirt.vif.LibvirtHybridOVSBridgeDriver
openstack-config --set /etc/nova/nova.conf DEFAULT \
network_api_class nova.network.neutronv2.api.API
```

```

openstack-config --set /etc/nova/nova.conf DEFAULT rpc_backend rabbit
openstack-config --set /etc/nova/nova.conf DEFAULT verbose True
openstack-config --set /etc/nova/nova.conf DEFAULT debug True
openstack-config --set /etc/nova/nova.conf neutron \
service_metadata_proxy True
openstack-config --set /etc/nova/nova.conf neutron \
metadata_proxy_shared_secret $metadata_shared_secret
openstack-config --set /etc/nova/nova.conf glance host $vip_glance
openstack-config --set /etc/nova/nova.conf neutron \
url http://$vip_neutron:9696/
openstack-config --set /etc/nova/nova.conf neutron \
admin_tenant_name services
openstack-config --set /etc/nova/nova.conf neutron \
admin_username neutron
openstack-config --set /etc/nova/nova.conf neutron \
admin_password neutron
openstack-config --set /etc/nova/nova.conf neutron \
admin_auth_url http://$vip_keystone:35357/v2.0
openstack-config --set /etc/nova/nova.conf conductor use_local false
openstack-config --set /etc/nova/nova.conf database \
connection mysql://nova:nova@$vip_db/nova
openstack-config --set /etc/nova/nova.conf database max_retries -1
//对于A/A高可用模式的scheduler, 需要新增以下配置
openstack-config --set /etc/nova/nova.conf DEFAULT \
scheduler_host_subset_size 30
openstack-config --set /etc/nova/api-paste.ini filter:authtoken\
auth_host $vip_keystone
openstack-config --set /etc/nova/api-paste.ini filter:authtoken\
admin_tenant_name services
openstack-config --set /etc/nova/api-paste.ini filter:authtoken\
admin_user nova
openstack-config --set /etc/nova/api-paste.ini filter:authtoken\
admin_password nova

```

Nova 相关配置完成后, 对 MariaDB 中的 nova 数据库进行授权, 并对 nova 数据库进行同步初始化以创建 Nova 服务运行所需的各种数据表, 参考如下:

```

//用户授权, 用户nova可以从任意位置访问数据库nova, 并对nova中的所有表有最高权限
mysql -uroot -proot -e "GRANT ALL PRIVILEGES ON nova.* TO \
'nova'@'localhost' IDENTIFIED BY 'nova';"
mysql -uroot -proot -e "GRANT ALL PRIVILEGES ON nova.* TO \
'nova'@'%' IDENTIFIED BY 'nova';"
//同步初始化nova数据库
su nova -s /bin/sh -c "nova-manage db sync"

```

Nova 配置完成和数据库准备就绪后, 在 Pacemaker 集群中创建 Nova 控制节点服务资源, 资源创建完成后, Pacemaker 将自动启动相关的 Nova 服务, 资源创建过程如下:

```

//Nova Pacemaker资源创建
pcs resource create nova-api systemd:openstack-nova-api \
--clone interleave=true

```



```

pcs resource create nova-scheduler systemd:openstack-nova-scheduler\
--clone interleave=true
pcs resource create nova-conductor systemd:openstack-nova-conductor\
--clone interleave=true
pcs resource create nova-cert systemd:openstack-nova-cert \
--clone interleave=true
pcs resource create nova-consoleauth systemd:openstack-nova-\
consoleauth --clone interleave=true
pcs resource create nova-novncproxy systemd:openstack-nova-\
novncproxy --clone interleave=true
//Nova资源启动顺序约束
pcs constraint order start keystone-clone then nova-consoleauth-clone
pcs constraint order start nova-consoleauth-clone then \
nova-novncproxy-clone
pcs constraint order start nova-novncproxy-clone then nova-api-clone
pcs constraint order start nova-api-clone then nova-cert-clone
pcs constraint order start nova-api-clone then nova-scheduler-clone
pcs constraint order start nova-scheduler-clone then \
nova-conductor-clone
//Nova资源节点绑定约束
pcs constraint colocation add nova-novncproxy-clone with \
nova-consoleauth-clone
pcs constraint colocation add nova-conductor-clone with \
nova-scheduler-clone
pcs constraint colocation add nova-scheduler-clone with nova-api-clone
pcs constraint colocation add nova-api-clone with nova-novncproxy-clone
pcs constraint colocation add nova-cert-clone with nova-api-clone

```

Nova 资源创建完成后，Pacemaker 将按照指定的 Order 约束和 Colocation 约束启动 Nova 相关资源服务，Pacemaker 资源启动完成后，通过 pcs resource 命令即可看到 Pacemaker 集群中 Nova 资源的运行情况，如下：

```

[root@controller1-vm ~]# pcs resource
.....
Clone Set: nova-api-clone [nova-api]
    Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: nova-scheduler-clone [nova-scheduler]
    Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: nova-conductor-clone [nova-conductor]
    Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: nova-consoleauth-clone [nova-consoleauth]
    Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: nova-novncproxy-clone [nova-novncproxy]
    Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: nova-cert-clone [nova-cert]
    Started: [ controller1-vm controller2-vm controller3-vm ]

```

可以看到，Pacemaker 已经启动所有用户创建的 Nova 资源，并且全部 Nova 资源均以 A/A 高可用模式运行在三个控制节点上。为了验证 Nova 服务在三个 OpenStack 控制节点上的运行情况，可以通过 Nova 命令行客户端查看 Nova 服务运行情况，结果如下：

```
[root@controller1-vm ~]source adminrc
[root@controller1-vm ~(keystone_admin)]$ openstack-status
== Nova services ==
openstack-nova-api:           active      (disabled on boot)
openstack-nova-cert:          active      (disabled on boot)
openstack-nova-compute:       inactive    (disabled on boot)
openstack-nova-network:       inactive    (disabled on boot)
openstack-nova-scheduler:     active      (disabled on boot)
openstack-nova-conductor:     active      (disabled on boot)
.....
```

```
[root@controller1-vm ~(keystone_admin)]$ nova service-list
+-----+-----+-----+-----+-----+-----+
| Id | Binary | Host | Zone | Status | State |
+-----+-----+-----+-----+-----+-----+
| 1 | nova-consoleauth | controller3-vm | internal | enabled | up |
| 3 | nova-consoleauth | controller1-vm | internal | enabled | up |
| 5 | nova-consoleauth | controller2-vm | internal | enabled | up |
| 6 | nova-cert | controller1-vm | internal | enabled | up |
| 8 | nova-scheduler | controller1-vm | internal | enabled | up |
| 11 | nova-cert | controller3-vm | internal | enabled | up |
| 14 | nova-scheduler | controller3-vm | internal | enabled | up |
| 16 | nova-conductor | controller1-vm | internal | enabled | up |
| 22 | nova-conductor | controller3-vm | internal | enabled | up |
| 25 | nova-scheduler | controller2-vm | internal | enabled | up |
| 26 | nova-cert | controller2-vm | internal | enabled | up |
| 27 | nova-conductor | controller2-vm | internal | enabled | up |
+-----+-----+-----+-----+-----+-----+
```

从 Nova 命令行客户端可以看到, Nova 控制节点服务已经被 Pacemaker 启动并正常运行在三个控制节点上(作为 Pacemaker 资源的服务必须由 Pacemaker 启停, 不要设置为开机自启动或手工启动)。Nova 控制节点服务正常运行后, 可以通过 Nova 添加虚拟机密钥对(Keypair)并进行安全组(SecurityGroup)的设置, 命令参考如下:

```
//添加Keypair
nova keypair-add admin-key
//设置安全组规则
nova secgroup-add-rule default icmp -1 -1 0.0.0.0/0
nova secgroup-add-rule default tcp 22 22 0.0.0.0/0
```

Keypair 和 SecurityGroup 设置完成后, 结果查看如下:

```
[root@controller1-vm ~(keystone_admin)]$ nova keypair-list
+-----+-----+-----+
| Name | Fingerprint |
+-----+-----+-----+
| admin-key | 43:cd:be:44:17:30:60:c1:4b:fd:4c:1a:d9:0e:29:13 |
+-----+-----+-----+
[root@controller1-vm ~(keystone_admin)]$ nova secgroup-list-rules\
default
+-----+-----+-----+-----+-----+-----+
| Id | Rule | Direction | EtherType | PortRange | Priority |
+-----+-----+-----+-----+-----+-----+
| 1 | icmp | ingress | 1 | -1:-1 | 1 |
| 2 | tcp | ingress | 6 | 22:22 | 1 |
+-----+-----+-----+-----+-----+-----+
```

IP Protocol	From Port	To Port	IP Range	Source Group
icmp	-1	-1	0.0.0.0/0	
tcp	22	22	0.0.0.0/0	

至此，Nova 控制节点高可用服务已经部署完成，全部 Nova 控制节点服务已经由 Pacemaker 集群资源管理器自动管理，并且可以提供正常的 Nova 交互控制服务。需要指出的是，本节并没有在控制节点中部署 Nova-compute 计算服务，Nova-compute 服务将会以高可用模式部署在独立的计算节点集群上，部署过程在后续将会进行介绍。本节介绍的 Nova API 高可用部署源代码可参考笔者位于 Github 上的开源项目（<https://github.com/ynwssjx/Openstack-HA-Deployment>），具体的部署脚本为 `11_create_nova_resource_on_pacemaker.sh`，执行该脚本后，将会在 Pacemaker 集群中创建高可用的 Nova API 服务。接下来介绍为 Openstack 提供监控和计量数据依据的 Ceilometer 数据采集服务高可用部署。

12.1.6 Ceilometer 数据采集服务高可用部署

Ceilometer 是 OpenStack 项目中的数据采集服务，能把 OpenStack 内部发生的几乎所有事件收集起来，然后为计费 and 监控以及其他服务提供数据支撑。Ceilometer 所采集的数据范围主要包括虚拟机实例性能数据（如 CPU、磁盘 IO 和网络 IO 等）以及 Cinder、Glance 和 Neutron 等服务项目的资源使用情况，Ceilometer 通过对 OpenStack 服务项目的数据收集，从而为上层的计费、结算或者监控应用提供统一的资源使用数据依据。Ceilometer 是一个相对较新的项目，其最初随 OpenStack 的 Folsom 版本发行，随后功能不断得到完善，尽管官方默认 Ceilometer 后端存储为 MongoDB，但是 Ceilometer 支持 DB2、HBase 和 SQLAlchemy 等后端数据库存储。Ceilometer 在 OpenStack 概念架构设计中的位置如图 12-6 所示，其与 Keystone 和 Horizon 项目类似，主要属于 OpenStack 云计算架构的资源治理层（Governance）和呈现（Presentation）层。

Ceilometer 的各个服务中，与采集相关的服务是 Ceilometer-collector、Ceilometer-agent-central、Ceilometer-agent-compute、Ceilometer-agent-notification。Ceilometer 的 Agent 负责数据采集，采集到的数据通过 RPC、UDP 或 File 方式 Publish 出去，具体的 Publish 方式通过 Pipeline 控制。在 Ceilometer 的数据采集 Agent 中，Agent-notification 负责收集各个组件推送到消息队列框架 oslo-messaging 的消息，Agent-compute 仅负责收集计算节点虚拟机的 CPU、内存和 IO 等信息，Agent-compute 仅部署在 OpenStack 计算节点上，Agent-central 通过各个组件的 API 进行有用数据的采集。Ceilometer 的数据采集方式主要分为两种，即 Poll 轮询和主动监听。Agent-compute 和 Agent-central 采用的是 Poll 轮询监听，即需要定期 Poll 轮询以采集信息，而 Agent-notification 为主动监听方式，其只需监听 AMQP 中的 Queue 即可采集到信息，Ceilometer 数据采集过程如图 12-7 所示。Ceilometer 的 Agent 采集到的数据可以汇聚到 Ceilometer-collector，再由 Ceilometer-collector 持久化保存在后端数

数据库，或者直接以 Publish 的方式传递给外部的监控或计费等应用程序，此外，外部应用程序还可以通过 Ceilometer 的 API 从数据库中访问 Ceilometer 采集到的数据。

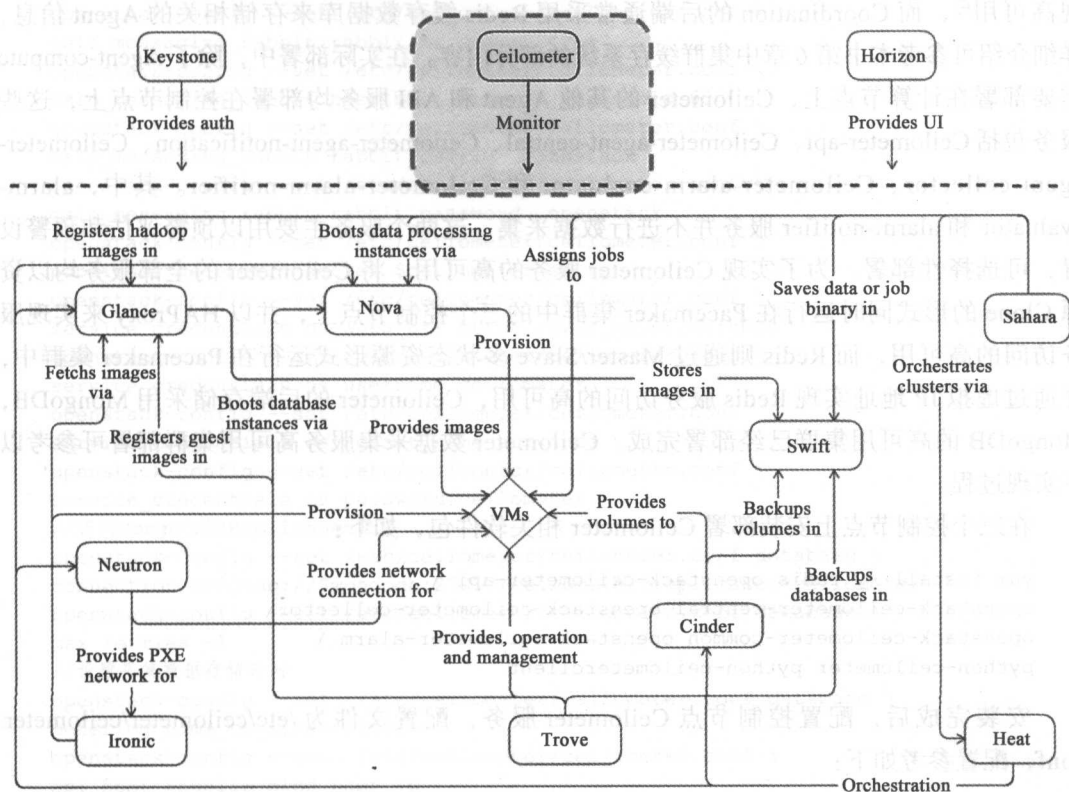


图 12-6 Ceilometer 在 OpenStack 概念架构设计中的位置

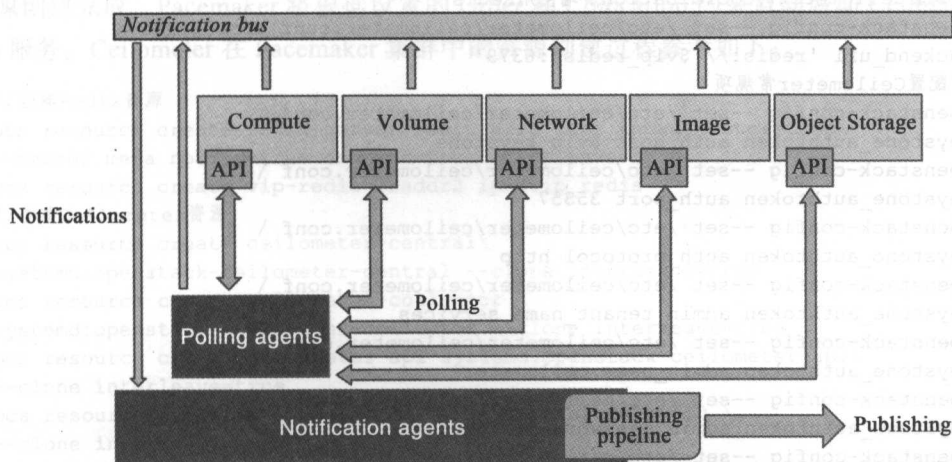


图 12-7 Ceilometer 数据采集过程

在 Ceilometer 的高可用部署中，需要用到协调组（Coordination Group）的概念。即分布在不同控制节点上的 Ceilometer Agent 之间需要通过 Coordination Group 的协调机制以实现高可用^①，而 Coordination 的后端通常采用 Redis 缓存数据库来存储相关的 Agent 信息，详细介绍可参考本书第 6 章中集群缓存系统的部分内容。在实际部署中，除了 Agent-compute 需要部署在计算节点上，Ceilometer 的其他 Agent 和 API 服务均部署在控制节点上，这些服务包括 Ceilometer-api、Ceilometer-agent-central、Ceilometer-agent-notification、Ceilometer-agent-collector、Ceilometer-alarm-evaluator 和 Ceilometer-alarm-notifier。其中，alarm-evaluator 和 alarm-notifier 服务并不进行数据采集，这两个服务主要用以预警评估和预警设置，可选择性部署。为了实现 Ceilometer 服务的高可用，将 Ceilometer 的全部服务均以资源 Clone 的形式同时运行在 Pacemaker 集群中的三个控制节点上，并以 HAProxy 来实现服务访问的高可用，而 Redis 则通过 Master/Slave 多状态资源形式运行在 Pacemaker 集群中，并通过虚拟 IP 地址实现 Redis 服务访问的高可用，Ceilometer 的后端存储采用 MongoDB，MongoDB 的高可用集群已经部署完成。Ceilometer 数据采集服务高可用集群部署可参考以下实现过程。

在三个控制节点上安装部署 Ceilometer 相关软件包，如下：

```
yum install -y redis openstack-ceilometer-api \
openstack-ceilometer-central openstack-ceilometer-collector \
openstack-ceilometer-common openstack-ceilometer-alarm \
python-ceilometer python-ceilometerclient
```

安装完成后，配置控制节点 Ceilometer 服务，配置文件为 /etc/ceilometer/ceilometer.conf，配置参考如下：

```
//配置Redis使其监听全部IP地址
sed -i "s/\s*bind \(.*)\$/#bind \1/" /etc/redis.conf
//配置Ceilometer的coordination后端使用Redis
openstack-config --set /etc/ceilometer/ceilometer.conf coordination
backend_url 'redis://'$vip_redis':6379'
//配置Ceilometer常规项
openstack-config --set /etc/ceilometer/ceilometer.conf \
keystone_authtoken auth_host $vip_keystone
openstack-config --set /etc/ceilometer/ceilometer.conf \
keystone_authtoken auth_port 35357
openstack-config --set /etc/ceilometer/ceilometer.conf \
keystone_authtoken auth_protocol http
openstack-config --set /etc/ceilometer/ceilometer.conf \
keystone_authtoken admin_tenant_name services
openstack-config --set /etc/ceilometer/ceilometer.conf \
keystone_authtoken admin_user ceilometer
openstack-config --set /etc/ceilometer/ceilometer.conf \
keystone_authtoken admin_password ceilometer
openstack-config --set /etc/ceilometer/ceilometer.conf DEFAULT \
```

① 如果未配置 Coordination，则仅能运行一个 agent-compute 和 agent-central 实例进程。

```

memcache_servers $memcaches
openstack-config --set /etc/ceilometer/ceilometer.conf \
oslo_messaging_rabbit rabbit_hosts $rabbitmq_hosts
openstack-config --set /etc/ceilometer/ceilometer.conf \
oslo_messaging_rabbit rabbit_ha_queues true
openstack-config --set /etc/ceilometer/ceilometer.conf \
oslo_messaging_rabbit heartbeat_timeout_threshold 60
openstack-config --set /etc/ceilometer/ceilometer.conf \
oslo_messaging_rabbit rabbit_userid openstack
openstack-config --set /etc/ceilometer/ceilometer.conf \
oslo_messaging_rabbit rabbit_password openstack
openstack-config --set /etc/ceilometer/ceilometer.conf \
publisher telemetry_secret $ceilometer_secret
openstack-config --set /etc/ceilometer/ceilometer.conf \
service_credentials os_auth_url http://$vip_keystone:5000
openstack-config --set /etc/ceilometer/ceilometer.conf \
service_credentials os_username ceilometer
openstack-config --set /etc/ceilometer/ceilometer.conf \
service_credentials os_tenant_name services
openstack-config --set /etc/ceilometer/ceilometer.conf \
service_credentials os_password ceilometer
//访问MongoDB的ReplicaSet集群
openstack-config --set /etc/ceilometer/ceilometer.conf database \
connection mongodb://$mongodb:27017/ceilometer?replicaSet=ceilometer
openstack-config --set /etc/ceilometer/ceilometer.conf database \
max_retries -1
//设置采集数据存储时间
openstack-config --set /etc/ceilometer/ceilometer.conf database \
metering_time_to_live 432000
openstack-config --set /etc/ceilometer/ceilometer.conf \
api host $public_bind_host_ip

```

Ceilometer 配置完成后，在 Pacemaker 集群中创建对应的 Redis 和 Ceilometer 资源，一旦资源创建完成，Pacemaker 将根据设置的 Order 和 Colocation 约束自动启动 Ceilometer 和 Redis 服务。Ceilometer 在 Pacemaker 集群中的资源创建过程参考如下：

```

//创建Redis资源
pcs resource create redis-server redis wait_last_known_master=true \
--master meta notify=true ordered=true interleave=true
pcs resource create vip-redis IPAddr2 ip=$vip_redis
//创建Ceilometer资源
pcs resource create ceilometer-central \
systemd:openstack-ceilometer-central --clone interleave=true
pcs resource create ceilometer-collector \
systemd:openstack-ceilometer-collector --clone interleave=true
pcs resource create ceilometer-api systemd:openstack-ceilometer-api \
--clone interleave=true
pcs resource create ceilometer-delay Delay startdelay=10 \
--clone interleave=true
pcs resource create ceilometer-alarm-evaluator \
systemd:openstack-ceilometer-alarm-evaluator --clone interleave=true
pcs resource create ceilometer-alarm-notifier \

```



```

systemd:openstack-ceilometer-alarm-notifier --clone interleave=true
pcs resource create ceilometer-notification \
systemd:openstack-ceilometer-notification --clone interleave=true
//设置Order启动顺序约束
pcs constraint order start mongodb-clone then ceilometer-central-clone
pcs constraint order start keystone-clone then ceilometer-central-clone
pcs constraint order promote redis-server-master then start vip-redis
pcs constraint order start vip-redis then ceilometer-central-clone \
kind=Optional
pcs constraint order start ceilometer-central-clone then \
ceilometer-collector-clone
pcs constraint order start ceilometer-collector-clone then \
ceilometer-api-clone
pcs constraint order start ceilometer-api-clone then \
ceilometer-delay-clone
pcs constraint order start ceilometer-delay-clone then \
ceilometer-alarm-evaluator-clone
pcs constraint order start ceilometer-alarm-evaluator-clone then \
ceilometer-alarm-notifier-clone
pcs constraint order start ceilometer-alarm-notifier-clone then \
ceilometer-notification-clone
//设置Colocation绑定约束
pcs constraint colocation add vip-redis with master redis-server-master
pcs constraint colocation add ceilometer-api-clone with \
ceilometer-collector-clone
pcs constraint colocation add ceilometer-delay-clone with \
ceilometer-api-clone
pcs constraint colocation add ceilometer-alarm-evaluator-clone with \
ceilometer-delay-clone
pcs constraint colocation add ceilometer-alarm-notifier-clone with \
ceilometer-alarm-evaluator-clone
pcs constraint colocation add ceilometer-notification-clone with \
ceilometer-alarm-notifier-clone

```

通常资源被添加到 Pacemaker 集群后会立即被启动，上述 `ceilometer-delay` 资源并非 Ceilometer 服务，而是为了在 Ceilometer-api 资源和 Ceilometer-alarm-evaluator 资源之间引入启动延时而添加的用户资源。Ceilometer 资源在 Pacemaker 集群中创建完成后，需要设置各个资源之间的启动依赖关系，依赖关系通过 Order 和 Colocation 约束实现。资源创建完成后，等待一定时间后（具体取决于控制节点硬件资源情况），Ceilometer 资源即可启动完成，通过 Pacemaker 集群资源查看命令 `pcs resource` 可以查看集群资源运行情况，如下：

```

[root@controller1-vm ~]# pcs resource
.....
Master/Slave Set: redis-server-master [redis-server]
  Masters: [ controller1-vm ]
  Slaves: [ controller2-vm controller3-vm ]
vip-redis      (ocf::heartbeat:IPaddr2):      Started controller1-vm
Clone Set: ceilometer-central-clone [ceilometer-central]
  Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: ceilometer-collector-clone [ceilometer-collector]

```

```

Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: ceilometer-api-clone [ceilometer-api]
Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: ceilometer-delay-clone [ceilometer-delay]
Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: ceilometer-alarm-evaluator-clone
[ceilometer-alarm-evaluator]
Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: ceilometer-alarm-notifier-clone
[ceilometer-alarm-notifier]
Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: ceilometer-notification-clone [ceilometer-notification]
Started: [ controller1-vm controller2-vm controller3-vm ]

```

可以看到, Pacemaker 已成功启动 Ceilometer 资源以及 Redis 资源, 其中 Redis 资源以 Master/Slave 的多状态资源形式运行, 而 Ceilometer 资源全部以 A/A 模式运行在三个控制节点上, Redis 和 Ceilometer 服务通过虚拟 IP 地址和 HAProxy 负载均衡器实现了客户端访问请求的高可用。本节介绍的 Ceilometer 高可用部署源代码可参考笔者位于 Github 上的开源项目 (<https://github.com/ynwssjx/Openstack-HA-Deployment>), 具体的部署脚本为 12_create_ceilometer_resource_on_pacemaker.sh, 执行该脚本后, 将会在 Pacemaker 集群中创建高可用的 Ceilometer 服务。

12.1.7 Heat 编排服务高可用部署

Heat 是 OpenStack 中的编排服务项目, 是一套完整的业务流程平台, Heat 可以帮助用户轻松实现和配置以 OpenStack 为基础资源的云服务体系, Heat 并非 OpenStack 云计算功能实现的核心模块, 但是 Heat 可以帮助云管理员和用户极大提高基于云平台的运行管理工作效率。编排 (Orchestration) 是一个与自动化运维或 DevOps 密切相关的术语, 即对已知事务按照预设流程依次排列自动执行, 从而实现预期目标。对传统 IT 数据中心而言, 编排即是服务器上架、CPU 安装、内存安装、硬盘安装、上电、安装配置网络接口、安装操作系统、配置操作系统、安装配置中间件、安装应用程序、配置应用发布程序等一整套安装配置过程。按照云计算的三层架构, 编排可以分为 IaaS 层编排、SaaS 层编排和 PaaS 层编排, 而 OpenStack 中的 Heat 主要服务于 IaaS 层的编排, 即虚拟机及其所需资源和操作系统层次的编排。在实际应用中, Heat 采用模板形式来定义和设计编排, 目前 Heat 支持业界主流的编排模板格式, 同时 Heat 还提供了大量模板示例供客户使用。Heat 同时支持基于 JSON 格式的 CFN 模板和基于 YAML 格式的 HOT 模板。CFN 模板主要是为了保持对 AWS 的兼容性, 而 HOT 模板是 Heat 自带的格式模板, 其资源类型更为丰富, 也更能体现出 Heat 模板自身的特点。

Heat 可以从不同方面进行资源编排, 如针对计算、网络和存储等资源所进行的 OpenStack 基础架构资源编排, 针对系统软件进行配置和部署的编排, 以及针对资源自动伸缩和负载均衡的高级编排。此外, 随着 DevOps 技术的流行, Heat 同 Chef、Puppet 和 Ansible

等自动化运维工具的结合也是 Heat 编排功能的发展方向。在 OpenStack 的虚拟机 HA 设计中，基于 Heat 模板栈和 Ceilometer 监控实现的实例 HA 方案也有不少用户在使用^①，Heat 模板高可用方案主要针对三个层次的进行高可用设计，即 Service、Instance 和 Stack 级别，其实现过程就是，如果虚机上的 Services 故障，则尝试重启服务，如果问题依旧则重启虚拟机，虚拟机重启之后仍然存在问题则重建整个 Stack。从解决问题的思路上看，Heat 高可用机制最终可以通过重建整个资源栈来彻底解决故障问题，这种基于重建 Stack 而解决单点故障的方案也是目前 OpenStack 社区实现 Nova 虚拟机高可用的主要方案之一，不过相对基于 Pacemaker 集群的高可用方案，Heat 高可用模板的设计和定义实现并不容易。

Heat 项目主要由 Heat-api、Heat-api-cfg 和 Heat-engine 服务组成。其中，Heat-api 是 OpenStack 中 Heat 项目的原生 API，主要负责将客户端请求转发给 Engine 进行处理，而 Heat-api-cfg 是 AWS 兼容的 Heat API，其主要作用是将 AWS 风格的请求转发给 Engine 进行处理，而 Heat-engine 则负责提供 Heat 最主要的协助功能。此外，Heat 还提供了 Heat-api-cloudwatch 服务，Cloudwatch 是 AWS 一项针对 AWS 云资源和在 AWS 上运行的应用程序进行监控的服务，目前 Heat 项目也支持部分 Cloudwatch 功能。Heat 在 OpenStack 概念架构设计中的层次位置如图 12-8 所示。

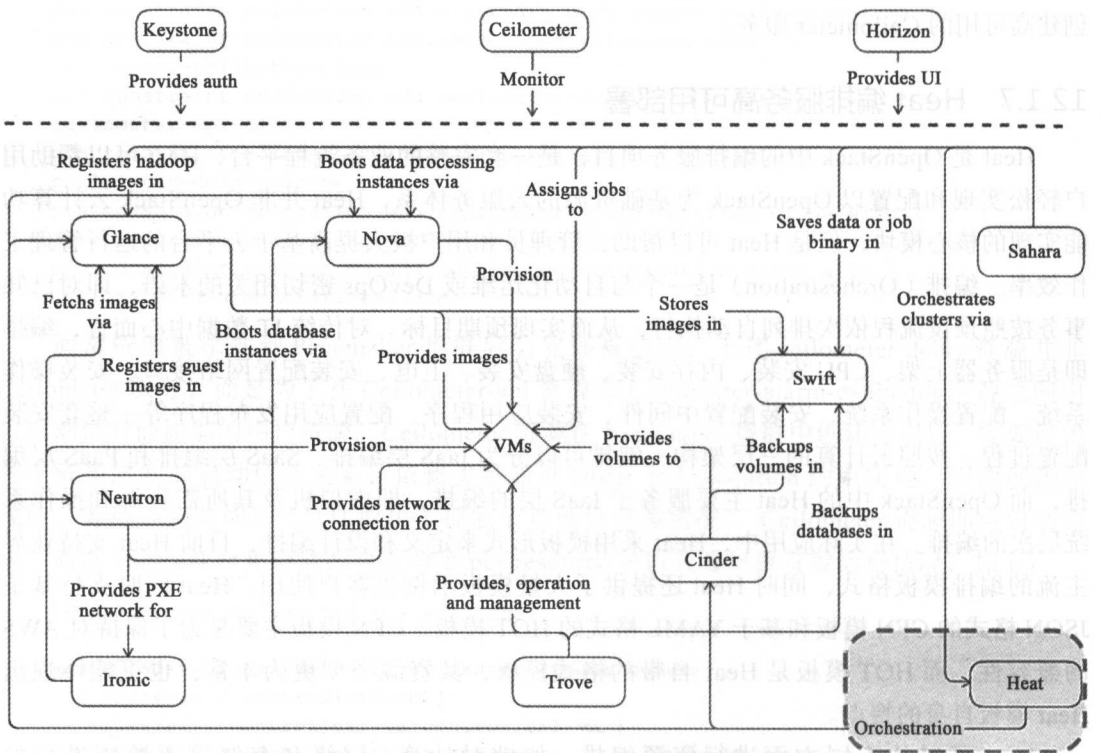


图 12-8 Heat 在 OpenStack 概念架构设计中的位置

① <https://wiki.openstack.org/wiki/Heat/HA>

在实际操作中，用户在 Horizon 界面或者命令行中提交包含模板和参数输入的请求，Horizon 或者命令行工具会把请求转化为 REST 格式的 API 调用，然后调用 Heat-api 或者 Heat-api-cfn。Heat-api 和 Heat-api-cfn 会验证模板的正确性，然后通过 AMQP 异步消息机制将请求传递给 Heat-Engine 进行处理。在基于 Pacemaker 集群的高可用部署中，Heat API 和 Engine 均以 Clone 资源形式运行在三个控制节点上，并通过虚拟 IP 地址和 HAProxy 负载均衡实现高可用，Heat 服务项目的高可用部署可参考如下实现过程。

安装配置 Heat 相关软件包，如下：

```
yum install -y openstack-heat-engine openstack-heat-api \
openstack-heat-api-cfn openstack-heat-api-cloudwatch \
python-heatclient openstack-utils python-glanceclient
```

配置 Heat 服务项目，配置文件为 /etc/heat/heat.conf，配置过程参考如下：

```
openstack-config --set /etc/heat/heat.conf database connection \
mysql://heat:heat@$vip_db/heat
openstack-config --set /etc/heat/heat.conf database database \
max_retries -1
openstack-config --set /etc/heat/heat.conf keystone_auth token \
admin_tenant_name services
openstack-config --set /etc/heat/heat.conf keystone_auth token \
admin_user heat
openstack-config --set /etc/heat/heat.conf keystone_auth token \
admin_password heat
openstack-config --set /etc/heat/heat.conf keystone_auth token \
service_host $vip_keystone
openstack-config --set /etc/heat/heat.conf keystone_auth token \
auth_host $vip_keystone
openstack-config --set /etc/heat/heat.conf keystone_auth token \
auth_uri http://$vip_keystone:35357
openstack-config --set /etc/heat/heat.conf keystone_auth token \
keystone_ec2_uri http://$vip_keystone:35357
openstack-config --set /etc/heat/heat.conf ec2_auth token \
auth_uri http://$vip_keystone:5000
openstack-config --set /etc/heat/heat.conf DEFAULT \
memcache_servers $memcaches
openstack-config --set /etc/heat/heat.conf oslo_messaging_rabbit \
rabbit_hosts $rabbitmq_hosts
openstack-config --set /etc/heat/heat.conf oslo_messaging_rabbit \
rabbit_ha_queues true
openstack-config --set /etc/heat/heat.conf oslo_messaging_rabbit \
heartbeat_timeout_threshold 60
openstack-config --set /etc/heat/heat.conf oslo_messaging_rabbit \
rabbit_userid openstack
openstack-config --set /etc/heat/heat.conf oslo_messaging_rabbit \
rabbit_password openstack
openstack-config --set /etc/heat/heat.conf heat_api bind_host \
$public_bind_host_ip
```

```

openstack-config --set /etc/heat/heat.conf heat_api_cfn bind_host \
$public_bind_host_ip
openstack-config --set /etc/heat/heat.conf heat_api_cloudwatch \
bind_host $public_bind_host_ip
openstack-config --set /etc/heat/heat.conf DEFAULT \
heat_metadata_server_url $vip_heat:8000
openstack-config --set /etc/heat/heat.conf DEFAULT \
heat_waitcondition_server_url $vip_heat:8000/v1/waitcondition
openstack-config --set /etc/heat/heat.conf DEFAULT \
heat_watch_server_url $vip_heat:8003
openstack-config --set /etc/heat/heat.conf DEFAULT rpc_backend \
heat.openstack.common.rpc.impl_kombu
openstack-config --set /etc/heat/heat.conf DEFAULT \
notification_driver heat.openstack.common.notifier.rpc_notifier
openstack-config --set /etc/heat/heat.conf DEFAULT \
enable_cloud_watch_lite false

```

设置 heat 用户对 MariaDB 中 heat 数据库的访问操作权限，同时同步 heat 数据库以创建 heat 数据库的初始数据表，如下：

```

GRANT ALL PRIVILEGES ON heat.* TO 'heat'@'localhost' IDENTIFIED BY '\
heat';
GRANT ALL PRIVILEGES ON heat.* TO 'heat'@'%' IDENTIFIED BY 'heat';
su heat -s /bin/sh -c "heat-manage db_sync"

```

Heat 配置完成和数据库准备就绪后，在 Pacemaker 集群中创建 Heat 相关资源，并为资源设置 Order 和 Colocation 约束，Heat 资源创建过程参考如下：

```

//创建Heat资源
pcs resource create heat-api systemd:openstack-heat-api \
--clone interleave=true
pcs resource create heat-api-cfn systemd:openstack-heat-api-cfn \
--clone interleave=true
pcs resource create heat-api-cloudwatch \
systemd:openstack-heat-api-cloudwatch --clone interleave=true
pcs resource create heat-engine systemd:openstack-heat-engine \
--clone interleave=true
//设置Order启动顺序约束
pcs constraint order start ceilometer-notification-clone \
then heat-api-clone
pcs constraint order start heat-api-clone then heat-api-cfn-clone
pcs constraint order start heat-api-cfn-clone \
then heat-api-cloudwatch-clone
pcs constraint order start heat-api-cloudwatch-clone \
then heat-engine-clone
//设置Colocation绑定约束
pcs constraint colocation add heat-api-cfn-clone with heat-api-clone
pcs constraint colocation add heat-api-cloudwatch-clone with \
heat-api-cfn-clone
pcs constraint colocation add heat-engine-clone with \
heat-api-cloudwatch-clone

```


Heat 资源创建完成后, Pacemaker 将按照 Order 约束依次启动 Heat 相关服务, 服务启动完成后, 在 Pacemaker 集群中可以通过 pcs resource 进行查看, 如下:

```
[root@controller1-vm ~]# pcs resource
.....
Clone Set: heat-api-clone [heat-api]
    Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: heat-api-cfn-clone [heat-api-cfn]
    Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: heat-api-cloudwatch-clone [heat-api-cloudwatch]
    Started: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: heat-engine-clone [heat-engine]
    Started: [ controller1-vm controller2-vm controller3-vm ]
```

可以看到, Heat-api、Heat-api-cfg、Heat-api-cloudwatch 和 Heat-engine 均以 A/A 高可用模式同时运行在三个控制节点上。至此, Heat 服务项目高可用部署已经完成, 值得注意的是, 随着服务资源的增加, Pacemaker 集群 CIB 配置信息也在不断增加和复杂, Pacemaker 做一次资源调整所消耗的系统资源和时间也随之增加, 因此资源创建完成后, 可能需要等待一段时间 Pacemaker 集群中的资源才会处于完全可用状态 (时间长短与系统资源配置有关)。本节介绍的 Heat 高可用部署源代码可参考笔者位于 Github 上的开源项目 (<https://github.com/ynwssjx/Openstack-HA-Deployment>), 具体的部署脚本为 13_create_heat_resource_on_pacemaker.sh, 执行该脚本后, 将会在 Pacemaker 集群中创建高可用的 Heat 服务。

12.1.8 Horizon 控制面板服务高可用部署

Horizon 是 OpenStack 中用以管理、控制 OpenStack 服务的 Web 控制面板, 通过 Horizon 提供的 Dashboard, 用户可以通过图形化界面管理实例、存储、网络、镜像等云计算资源。此外, 用户还可以在控制面板中使用终端或 VNC 直接访问实例, 并查看实例的操作日志。总之, OpenStack 中通过各个项目的命令行可以实现的功能, 通过 Horizon 的 Dashboard 界面几乎都可以实现, 因此, Horizon 是 OpenStack 各个服务项目与用户进行交互的统一管理界面。Horizon 服务项目在 OpenStack 概念架构设计中层次位置如图 12-9 所示。

在实际应用中, Horizon 是进行二次开发最频繁的项目之一, 例如进行交互界面功能菜单的重新排版、背景/前景颜色更改、Logo 更换和功能按键的添加等都需要修改 Horizon 的源代码。Horizon 是基于 Django 开发的 Web 项目, 其代码主要分布在 Horizon 和 OpenStack-dashboard 子目录中, Horizon 下面主要放的是一些最基本的、可以共享的类以及表格和模板等, 而 OpenStack-Dashboard 下面主要存放的是跟最终的交互界面有直接关系或更加具体的类、表格和模板等, 这些对象即是用户进行二次开发所需要更改的地方。

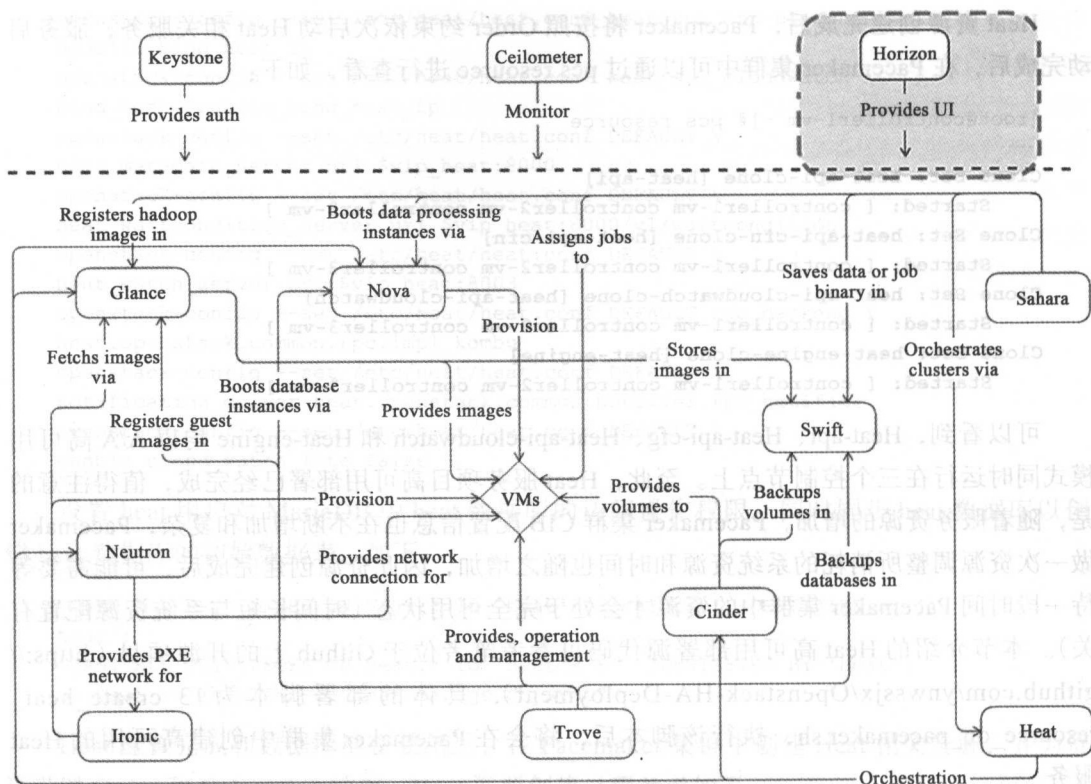


图 12-9 Horizon 在 OpenStack 概念设计架构中的位置

在部署时，Horizon 通常以 Apache Web Server 形式提供服务，与 Horizon 密切相关的 OpenStack 核心服务是认证服务 Keystone，用户可以选择 Horizon 是否支持其他服务项目的图形化界面管理，如 Nova、Neutron 和 Cinder 等，默认对 OpenStack 核心服务均支持。为了更好的用户体验，用户客户端浏览器必须支持 HTML5，并且需要启用 Cookies 和 Javascripts，如果希望使用 VNC 访问虚拟机，则浏览器还必须支持 HTML5Canvas 和 HTML5 WebSockets。由于 Horizon 以 Web Server 形式运行，因此其高可用模式配置较为简单，只需在三个控制节点上同时运行 Web Server，然后通过虚拟 IP 地址和 HAProxy 负载均衡器即可实现 Horizon 服务的高可用。Horizon 服务的高可用部署配置可参考以下实现。

安装配置与 Horizon 相关的软件，如下：

```
yum install -y mod_wsgi httpd mod_ssl openstack-dashboard memcached\
pythonmemcached
```

配置 Horizon 服务，主要配置文件为 `/etc/openstack-dashboard/local_settings`，Horizon 的服务地址及允许访问的主机等参数均在该文件中设置，配置过程可以参考以下代码段：

```
sed -i -e "s#ALLOWED_HOSTS.*#ALLOWED_HOSTS = ['*',]#g" \
-e "s#^CACHES#SESSION_ENGINE = \\"
```

```

'django.contrib.sessions.backends.cache'\nCACHES#g#" \
-e "s#locmem.LocMemCache'#memcached.MemcachedCache',\n\t'LOCATION':\
$horizonmemcachednodes#g" \
-e 's#OPENSTACK_HOST =.*#OPENSTACK_HOST = '$vip_keystone'#"#g' \
-e "s#^LOCAL_PATH.*#LOCAL_PATH = '/var/lib/openstack-dashboard'#g" \
-e "s#SECRET_KEY.*#SECRET_KEY = '$horizon_secret'#g#" \
/etc/openstack-dashboard/local_settings
echo "COMPRESS_OFFLINE = True" >> \
/etc/openstack-dashboard/local_settings
python /usr/share/openstack-dashboard/manage.py compress
//配置Apache仅监听80端口
sed -i -e "s/^Listen.*/Listen $public_bind_host_ip:80/g" \
/etc/httpd/conf/httpd.conf
//启用server-status, pacemaker将使用此功能以验证apache正在响应中
cat > /etc/httpd/conf.d/server-status.conf << EOF
<Location /server-status>
    SetHandler server-status
    Order deny,allow
    Deny from all
    Allow from localhost
</Location>
EOF

```

配置完成后, 在 Pacemaker 集群中创建 Horizon 资源, 因为 Horizon 资源以 Apache 服务形式运行, 因此创建 Horizon 资源实际上是创建 Apache 资源, 资源创建过程如下:

```
pcs resource create horizon apache --clone interleave=true
```

Horizon 资源创建完成后, Pacemaker 将自动以 Apache Web Server 形式启动 Horizon 服务, 启动之后, 通过 pcs resource 命令即可看到 Pacemaker 中的全部资源运行情况, 如下:

```
[root@controller1-vm ~]# pcs resource
```

```
.....
```

```
Clone Set: horizon-clone [horizon]
```

```
Started: [ controller1-vm controller2-vm controller3-vm ]
```

可以看到, Horizon 资源已经在 Pacemaker 集群中启动, 并正常运行在三个控制节点上。现在, 通过 Web 浏览器即可进入 OpenStack 的 Dashboard 控制面板中, 本例中 Horizon 控制面板服务的虚拟 IP 地址为 192.168.142.211, 浏览器输入 <http://192.168.142.211/dashboard> 即可访问 OpenStack 控制面板服务, 如图 12-10 所示。

本节介绍的 Horizon 高可用部署源代码可参考笔者位于 Github 上的开源项目 (<https://github.com/ynwssjx/Openstack-HA-Deployment>), 具体的部署脚本为 14_create_horizon_resource_on_pacemaker.sh, 执行该脚本后, 将会在 Pacemaker 集群中创建高可用的 Horizon 服务。截至目前, OpenStack 高可用集群控制节点服务已经部署完成, 控制节点服务包括 OpenStack 依赖基础软件服务和 OpenStack 核心项目服务, 其中基础软件服务包括数据库服务、消息队列服务、负载均衡服务和缓存系统服务, OpenStack 核心项目服务包括 Keystone、Glance、Cinder、Nova、Neutron、Ceilometer、Heat 和 Horizon, 而这些部署在

三个控制节点上的服务已全部交由 Pacemaker 集群资源管理器控制，并通过 Pacemaker 和 HAProxy 负载均衡器的结合实现了服务的高可用。但是，到目前为止，OpenStack 云计算中提供虚拟机服务的计算节点 Nova-compute 及其相关服务仍然没有部署，下一节将重点介绍如何实现 Nova-compute 服务及其相关计算节点服务的高可用部署。

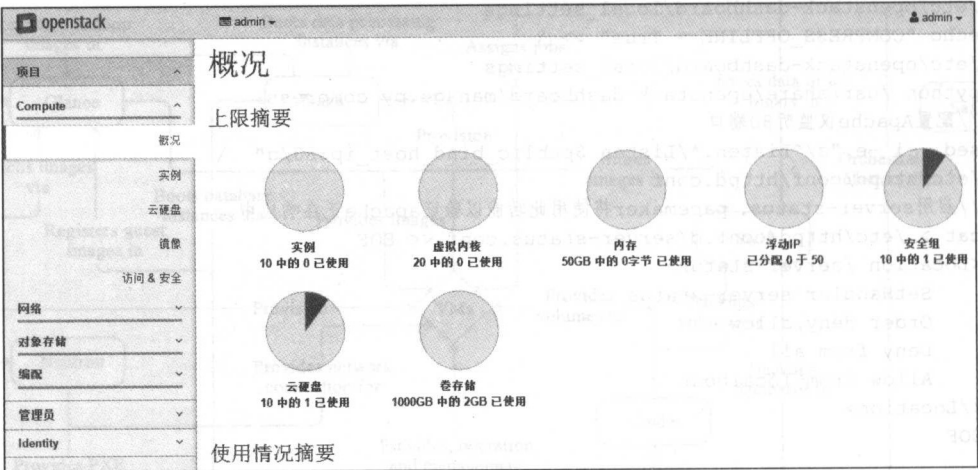


图 12-10 OpenStack Dashboard 控制面板

12.2 OpenStack 计算节点服务高可用部署

12.2.1 OpenStack 计算节点高可用实现概述

控制节点服务是 OpenStack 高可用集群的调度控制中心，而计算节点服务才是 OpenStack 高可用集群中真正实现云计算服务的核心。在 OpenStack 云计算中，虚拟机通常由独立的计算节点提供，并由 Nova 项目的 Nova-compute 服务负责实现，而计算节点高可用部署的焦点在于如何实现虚拟机实例的高可用。在理想的云计算环境中，每一个运行在 OpenStack 之上的业务负载都是云上的原生应用（Native Application），其应该具备水平扩展能力和应对任何可能的服务器宕机情况下的故障容忍（Fault Tolerant）能力，但是实际情况往往并非如此。随着云计算的普及和 OpenStack 的成熟，越来越多的用户开始基于 OpenStack 部署自己的私有云，并迫切地将已部署在虚拟化平台上的传统应用迁移到私有云中，而传统业务负载对于高可用服务器的依赖使得 OpenStack 社区对于实现虚拟机 HA 的呼声越来越高，因此 OpenStack 计算节点高可用的特性需求也越发迫切。但是，OpenStack 社区一直认为实例 HA 应该由上层应用软件实现，而不应该由 OpenStack 的核心功能提供。因此，社区并没有提供完整的实例 HA 解决方案，而仅提供了一些配合实现的外围监控服务和虚拟机撤离（Evacuate）机制，由于社区没有完整成熟的实例 HA 机制，因此用户必须根据自己的业务特点结合第三方软件设计和部署自己的高可用方案。

对于很多传统企业用户而言,固有的思维方式和程序设计模式均要求服务器具备高可用性,尤其是在虚拟化市场上,以 VMware 为主的虚拟化厂商充分满足了企业用户虚拟机高可用的需求,而且市场上主流的虚拟化平台为了满足客户需求均提供计算节点的高可用功能,从而保证用户计算节点故障时,虚拟机能够平滑迁移到其他计算节点上继续运行。因此很多传统企业应用对计算节点的高可用性非常依赖,而缺少完整的计算节点高可用解决方案也成为阻碍 OpenStack 进入很多企业用户的关键障碍。值得指出的是,OpenStack 社区也并非一成不变,为了配合用户实现虚拟机高可用,在 OpenStack 的 Liberty 和 Mitaka 版本中, Nova 项目的 API 也在针对性地做出改变以更好地实现对 Nova 服务状态改变的监控和虚拟机的迁移过程。而在 2016 年 Austin 峰会上, OpenStack 高可用社区已经在着手设计和开发官方统一的实例高可用解决方案^①,不过截至 Newton 版本发行,仍然没有完整的实例高可用解决方案可供用户直接部署使用,而官方推荐的实例高可用参考方案主要有 RedHat 所采用的 OCF 资源代理 (RAs) 方案^②、NTT 基于 Masakari 实现的实例高可用方案^③和 Intel 基于 Mistral 实现的虚拟机自动撤离高可用方案^④。在上述三种方案中, Redhat 基于 Pacemaker_remote 和 OCF RAs 的计算节点虚拟机高可用方案是相对较为成熟和参考使用较为普遍的实例高可用方案,也是本章所参考使用的计算节点高可用设计实现方案。

12.2.2 OpenStack 计算节点高可用方案分析

对于 OpenStack 实例 HA 方案而言,不论是 RedHat、NTT 或者 Intel 的方案,都离不开三个环节,即监控 (Monitoring)、隔离 (Fencing) 和恢复 (Recovery)。不同厂商和用户各自采用的不同实例 HA 方案,其主要的区别在于监控环节,故障监控是后续服务隔离和恢复的基础前提,因此所采用监控方案的准确性和效率高低直接影响了不同实例 HA 方案的优劣。在实例 HA 中,监控需要实现两个主要任务,即主机故障监控和触发对于主机故障的响应操作 (隔离和恢复),关于主机故障的监控,社区一直有“应该将其集成进 OpenStack,并由 Nova 来实现”的呼声,此外也有通过 Heat 或者其他 OpenStack 项目来实现的建议,不过社区主流的意见仍然认为监控功能不应该由 OpenStack 本身来实现,而应该是支撑 OpenStack 部署实现的基础架构软件完成的事情。所以,在故障监控这个问题上,社区不建议持续纠结是否应将其归入某个项目来实现,而是考虑通过支撑 OpenStack 部署实现的基础架构软件来实现,如很多用户已经在使用的 Pacemaker 集群管理软件便能很好实现此功能,虽然早期的 Pacemaker 集群有节点数目限制的缺陷,但是 Pacemaker 集群社区发布的 Pacemaker_remote 功能将计算节点作为 Pacemaker 集群的一部分,同时使其独立于 Corosync 集群,从而解决了 Pacemaker 集群节点数目限制的问题,因而大规模的计算节

① <http://docs.openstack.org/ha-guide/compute-node-ha.html>

② <http://aspiers.github.io/openstack-summit-2016-austin-compute-ha/#/ocf-pros-cons>

③ <https://launchpad.net/masakari>

④ <https://github.com/gryf/mistral-evacuate>

点集群也可以通过运行 `Pacemaker_remote` 加入 `Pacemaker` 控制集群并实现节点监控功能。除了 `Pacemaker` 之外，也有部分用户将 `Zookeeper` 和 `Nagios` 用于 `OpenStack` 的服务故障监控场景。

隔离是服务故障监控后的首要操作，隔离的主要目的在于将故障节点完全孤立隔离，通常的隔离操作是利用服务器上诸如 `IPMI` 的功能直接将服务器下电或重启。隔离操作对于一个高可用集群而言是十分重要的。一方面，节点故障的原因有多种，而集群在其他地方重新启动虚拟机之前，必须确保故障节点完全失去服务能力，以免同一个虚拟机在不同节点同时运行。另一方面，`OpenStack` 高可用集群在实现实例 HA 时经常采用基于共享存储的高可用机制，如果多个节点同时运行相同虚拟机，则会导致同时访问共享存储而损坏数据。此外，还有可能导致网络上出现两次相同的 IP 地址。因此，对于 `OpenStack` 高可用集群，集群管理器必须具备节点隔离功能，而在这方面，`Pacemaker` 具有天然优势，因为其内置集成了节点隔离功能，如果采用类似 `Nagios` 等集群监控工具，则需要额外考虑隔离功能的实现。

一旦集群监控到故障行为并且故障节点已被隔离，则下一步措施便是触发虚拟机撤离并在其他计算节点上恢复虚拟机。为了实现虚拟机的故障转移，`Nova` 提供了专门的 API 以将虚拟机由故障主机迁移至正常主机，同时为了完成虚拟机的迁移，用户实例系统镜像必须位于共享存储上，或者作为替代解决方案，也可以位于 `Cinder` 块存储卷上或 `Ceph RBD` 存储池中，虚拟机的恢复过程即故障发现和节点隔离后，通过一定的触发机制调用 `Nova` 相应的 API 以实现故障节点虚拟机迁移的过程。迁移完成后，虚拟机的系统镜像和应用程序并没有改变，仅是在其他节点上由相同的镜像重新启动虚拟机，而该虚拟机的全部属性（如 `UUID`）仍然保持不变。

12.2.3 OpenStack 计算节点 Pacemaker 高可用集群分析

在各种 `OpenStack` 实例 HA 方案中，本章采用的是基于 `Pacemaker/Pacemaker_remote` 的 `OCF RAs` 计算节点高可用实现方案。在此方案中，控制节点运行 `Pacemaker` 高可用集群，其中每个控制节点均运行 `Corosync` 集群全栈程序和全部 `Pacemaker` 组件，并构成 `OpenStack` 高可用集群的控制面。由于 `Corosync` 集群 16 节点的限制，计算节点并不运行 `Pacemaker` 全栈集群程序，而是运行取而代之的 `Pacemaker_remote`。`Pacemaker_remote` 允许计算节点加入 `Pacemaker` 集群，并将计算节点资源纳入 `Pacemaker` 集群资源管理器。`Pacemaker_remote` 可以认为是 `Pacemaker` 的精简版，同时是 `Pacemaker` 本地资源管理进程（`Local Resource Management Deamon, LRMD`）的增强版，其允许节点在未运行 `Corosync` 集群软件的前提下以节点资源形式加入 `Pacemaker` 集群，并成为 `Pacemaker` 集群成员节点，从而使其上运行的服务资源接受 `Pacemaker` 的统一管理和调度。由于 `Pacemaker` 集群 16 节点限制的缺陷本质上由集群 `Corosync` 消息层所引起，而并非 `Pacemaker` 组件程序造成，并且运行 `Pacemaker_remote` 的计算节点不计入 `Corosync` 集群，因此即使在较大规模计算节点

部署的情况下，也不会受到 Pacemaker 集群节点限制的约束（除非运行 Pacemaker 的控制节点数目超出 16 个）。在 Pacemaker 集群中，运行 Corosync 全栈软件和 Pacemaker 全部组件的节点称为 Pacemaker 集群节点（Cluster Node），而仅运行 Pacemaker_remote 的计算节点则称为远程节点（Remote Node），远程节点可以像集群节点一样运行 Pacemaker 集群资源和大部分命令行工具，但是不具备成为 Pacemaker 集群主指定控制器（Designated Controller, DC）节点的资格，不能参与 Quorum 投票，也不能在集群中发起 Fencing 操作^①。

在基于 Pacemaker 的 OpenStack 高可用集群中，运行全栈 Pacemaker 集群软件的控制节点称为 Cluster Node，而运行 Pacemaker_remote 的计算节点称为 Remote Node。从 Pacemaker 集群角度而言，由 OpenStack 控制节点组成的 Pacemaker 集群节点负责提供高可用集群的 DC、Quorum 和 Fencing 等集群全局控制功能，通常将其称为控制层面（Control Plane），而由 OpenStack 计算节点组成的 Pacemaker 远程节点则负责运行 OpenStack 计算服务并接受控制层面的调度控制，通常将其称为受控层面或计算层面（Compute Plane）。从 OpenStack 集群功能而言，Pacemaker 集群节点负责运行 OpenStack 服务组件的控制和调度等 API 服务，而 Pacemaker 远程节点负责提供 OpenStack 计算项目 Nova 的 Compute 服务。

12.2.4 OpenStack 计算节点 Pacemaker 高可用集群实现

在部署实现 OpenStack 计算节点 Pacemaker 高可用集群之前，必须确保 Pacemaker 控制层面已经部署完成，并且 OpenStack 控制节点高可用服务已在 Pacemaker 集群中正常运行。在正式部署计算节点 OpenStack 高可用服务之前，必须考虑以下几种情况：

- ❑ Pacemaker 集群节点正常运行，OpenStack 控制节点高可用服务正常运行；
- ❑ Pacemaker 集群控制层面已经启用节点 Fencing 功能；
- ❑ Pacemaker 集群节点（即 OpenStack 控制节点）和远程节点（即 OpenStack 计算节点）上已经安装满足如下要求的软件包：
 - fence-agents-all-4.0.11-27.el7_2.5.x86_64 或者更高版本；
 - pacemaker-1.1.13-10.el7_2.2.x86_64 或者更高版本；
 - resource-agents-3.9.5-54.el7_2.6.x86_64 或者更高版本。
- ❑ 集群环境中已经准备好供 Nova 虚拟机使用的共享临时存储和块存储；
- ❑ 控制层面和计算层面服务在高可用配置期间均需要暂时停止服务。

对于共享存储的需求，也可以有其他替代解决方案。对于 Nova 而言，默认从计算节点临时存储启动虚拟机时，实例镜像存放路径为 /var/lib/nova/instances 目录，因此为了实现实例 HA，该目录必须配置为共享目录。而如果从 Cinder 的块存储 Volume 中启动虚拟机，则实例镜像存储在 Volume 中无须额外配置共享存储存放实例镜像。除此之外，也可以配置 Ceph RBD 块存储作为 Nova 虚拟机临时存储后端，由于 Nova 实例镜像存储在 Ceph 存

^① 远程节点不能发起 Fencing 操作，但是只要其具备 Fencing 设备，则集群节点便可将其隔离。

储池中，因而也无须额外配置共享存储。需要注意的是，在计算节点安装满足上述版本要求的 Resource-agents 和 Fence-agents-all 之后，计算节点将会新增 NovaEvacuate 和 Fence_compute 资源代理，而在使用 NovaEvacuate OCF 资源代理创建 Pacemaker 资源时，默认使用的是共享存储模式，而如果未采用共享存储模式，则可以在资源创建时指定 no_shared_storage=1，否则 NovaEvacuate 资源代理将会出现 “InvalidSharedStorage” 的错误提示。OpenStack 计算节点高可用部署可参考如下过程实现。

首先需要准备计算节点操作系统环境，如 NTP 设置、防火墙设置、SELinux 设置、yum 源设置、NFS 配置、Pacemaker_remote 等集群软件安装和 Libvirt 虚拟化相关软件安装等，计算节点系统准备请参考本书 11.3.3 节。确认计算节点系统环境准备完成之后，在计算节点安装 OpenStack 计算服务相关软件包，如下：

```
yum install -y openstack-nova-compute openstack-neutron-openvswitch
openstack-ceilometer-compute openstack-utils python-cinder
python-memcached openstack-neutron
```

本节采用共享存储模式实现实例 HA，因此在 NFS 服务器上创建共享目录时，计算节点将会通过 NFS 服务将其挂载到本地 /var/lib/nova/instances 目录上，如下：

```
mkdir -p $share_dir/instances
chown nova:nova $share_dir/instances
```

这里的 \$share_dir 表示共享目录的上层父路径，如 /data。在计算节点上配置集群相关服务的启动情况，如下：

```
systemctl enable pcsd
systemctl start pcsd
systemctl enable openvswitch
systemctl start openvswitch
systemctl stop libvirtd
systemctl disable libvirtd
systemctl enable pacemaker_remote
systemctl start pacemaker_remote
```

创建 Pacemaker_remote 使用的授权 Key，在 Pacemaker 集群中，全部集群节点和远程节点都采用相同的 Key 文件。因此，秉承“一次创建，多次使用”的原则，将此处创建的 Key 文件分发到全部集群节点 /etc/pacemaker 目录中，如下：

```
mkdir -p /etc/pacemaker
dd if=/dev/urandom of=/etc/pacemaker/authkey bs=4096 count=1
scp /etc/pacemaker/authkey controller[1,2,3]-vm:/etc/pacemaker
scp /etc/pacemaker/authkey computer[1,2]:/etc/pacemaker
```

上述 controller[1, 2, 3]-vm 表示三个控制节点 controller1-vm、controller2-vm 和 controller3-vm，computer[1, 2] 表示两个计算节点 computer1 和 computer2。除了授权 Key 之外，还需为 Pacemaker 集群用户 hacluster 设置密码，密码与控制节点 hacluster 用户相

同，如下：

```
echo $hacluster_passwd | passwd --stdin hacluster
```

在开始集群配置之前，检查控制节点和计算节点 Linux 系统中是否已经安装符合版本要求的 Pacemaker (1.1.13-10.el7_2.2.x86_64)、Fence-agents (4.0.11-27.el7_2.5.x86_64) 和 Resource-agents (3.9.5-54.el7_2.6.x86_64) 软件包，如下：

```
# rpm -qa | egrep '(pacemaker|fence-agents-compute|resource-agents)'
pacemaker-libs-1.1.13-10.el7.x86_64
pacemaker-cli-1.1.13-10.el7.x86_64
pacemaker-1.1.13-10.el7.x86_64
resource-agents-3.9.5-54.el7.x86_64
fence-agents-compute-4.0.11-27.el7.x86_64
pacemaker-cluster-libs-1.1.13-10.el7.x86_64
pacemaker-remote-1.1.13-10.el7.x86_64
```

在计算节点上配置与其相关的 Openstack 服务，即 Nova-compute、Ceilometer-compute 和 Neutron-openvswitch 服务，计算节点 Openstack 服务的配置脚本可参考笔者位于 Github 上的开源项目 (<https://github.com/ynwssjx/Openstack-HA-Deployment>)，具体的配置脚本名称为 openstack_compute_node_nova_install_and_config.sh。计算节点相关的 OpenStack 服务配置完成后，即可在 Pacemaker 集群中配置 OpenStack 计算节点高可用资源（此过程在控制节点上操作）。这里需要解决的一个问题是，如何确保属于控制层面的 OpenStack 相关服务仅运行在控制节点（即 Pacemaker 集群节点）上，而属于计算层面的 OpenStack 相关服务仅运行在计算节点（即 Pacemaker 远程节点）上。为了解决这一问题，需要通过 Pacemaker 集群的属性（Property）为控制节点设置一个属性标签（如 controller），同时为计算节点也设置一个属性标签（如 computer），并通过资源的 Location 约束将属于不同层面的 Pacemaker 资源打上不同的节点标签（controller 或 computer），从而将其固定到各自节点上运行，如属于控制层面的资源仅在控制节点上运行，而属于计算层面的资源仅在计算节点上运行。Pacemaker 资源创建过程参考如下：

首先，创建 Active/Passive 模式运行的 Nova-evacuate 资源，该资源由 NovaEvacuate OCF 资源代理实现^①，并以 Active/Passive 模式运行在控制节点上，创建时需要提供 Keystone 服务的访问地址、租户名称以及用户名和密码，Nova-evacuate 资源创建过程如下：

```
pcs resource create nova-evacuate ocf:openstack:NovaEvacuate \
auth_url=http://$vip_keystone:35357 username=admin password=admin \
tenant_name=admin
```

Nova-evacuate 资源需要在虚拟 IP 地址资源、Glance、Nova 和 Neutron 服务之后启动，通过 Order 约束设置启动顺序，如下：

^① NovaEvacuate 为 Redhat 专门针对实例高可用而开发的 OCF 脚本。

```
//设置Nova-evacuate与虚拟IP地址之间的启动顺序
for i in vip-glance vip-cinder vip-neutron vip-nova vip-db vip-rabbitmq\
vip-keystone cinder-volume;
do
    pcs constraint order start $i then nova-evacuate
done
//设置Nova-evacuate与镜像、网络和计算服务之间的启动顺序
for i in glance-api-clone neutron-metadata-agent-clone \
nova-conductor-clone;
do
    pcs constraint order start $i then nova-evacuate require-all=false
done
```

进行后续操作之前，先停用控制层面与 OpenStack 相关的全部资源。由于 Keystone 是全部 OpenStack 资源启动的基础，因此只需停止 Keystone 资源即可，其他资源将会自动停止，如下：

```
pcs resource disable keystone --wait=480
```

资源停止的默认等待时间为 60min，等待时间可以根据全部需要停止资源的 timeout 参数值进行累加计算得到并以 --wait 参数指定（这里为 8min）。待控制层面全部 OpenStack 相关资源停止完成后，从 Pacemaker 集群中获取控制节点列表，并为控制节点打上名为“controller”的标签^①，以表明该节点为控制节点，且仅运行控制层面相关的资源，如下：

```
//获取控制节点列表
controllers=$(cibadmin -Q -o nodes | grep uname | sed s/.*/uname...// | \
awk -F\" '{print $1}' | grep -v comput*)
//设置控制节点标签
for controller in ${controllers}; do
    pcs property set --node ${controller} node_role=controller
done
```

为控制节点打上属性标签后，需要为 Pacemaker 中的资源设置 Location 约束，以使其仅运行在贴有“controller”标签的节点上。通过 cibadmin 可以一次性获取 Pacemaker 集群中的资源，但是需要将集群中的 Stonith 设备过滤之后再行 Location 设置，具体参考如下：

```
stonithdevs=$(pcs stonith | awk '{print $1}')
for i in $(cibadmin -Q --xpath //primitive --node-path | tr ' ' '\n' | \
awk -F"id=" '{print $2}' | awk -F'"' '{print $1}' | uniq);
do
    found=0
    if [ -n "$stonithdevs" ]; then
        for x in $stonithdevs; do
            if [ $x = $i ]; then
                found=1
            fi
        done
    fi
```

^① 标签名称仅为“望文生义”的方便，名称可自定义。

```

        fi
    done
fi
//对不属于stonith设备的资源进行Location约束设置,使其仅运行在控制节点上
if [ $found = 0 ]; then
    pcs constraint location $i rule resource-discovery=exclusive\
        score=0 node_role eq controller --force
fi
done

```

现在,可为计算层面创建 Pacemaker 资源(即运行在计算节点上的 OpenStack 服务)。由于计算节点还未加入 Pacemaker 集群,为了避免创建后的资源即刻在控制节点上运行,需在创建资源时加上 `--disabled` 和 `--force` 参数,以告知 Pacemaker 该资源创建后立刻处于停用状态(即不要启动此资源)。此外,通过 Location 约束为资源打上“computer”标签,这样后续计算节点加入 Pacemaker 集群并设置“computer”节点属性标签后,被贴上“computer”标签的资源将会仅运行在计算节点上。计算节点 Pacemaker 资源创建过程参考如下:

```

//创建计算节点neutron-openvswitch-agent资源
pcs resource create neutron-openvswitch-agent-compute \
systemd:neutron-openvswitch-agent --clone interleave=true \
--disabled --force
//设置Location约束,使其仅运行在计算节点上
pcs constraint location neutron-openvswitch-agent-compute-clone rule
resource-discovery=exclusive score=0 node_role eq compute
//设置启动顺序约束
pcs constraint order start neutron-server-api-clone then
neutron-openvswitch-agent-compute-clone require-all=false
//创建计算节点libvirtd资源
pcs resource create libvirtd-compute systemd:libvirtd --clone \
interleave=true --disabled --force
pcs constraint location libvirtd-compute-clone rule \
resource-discovery=exclusive score=0 node_role eq compute
pcs constraint order start neutron-openvswitch-agent-compute-clone\
then libvirtd-compute-clone
pcs constraint colocation add libvirtd-compute-clone with \
neutron-openvswitch-agent-compute-clone
//创建计算节点openstack-ceilometer-compute资源
pcs resource create ceilometer-compute \
systemd:openstack-ceilometer-compute --clone interleave=true \
--disabled --force
pcs constraint location ceilometer-compute-clone rule \
resource-discovery=exclusive score=0 node_role eq compute
pcs constraint order start ceilometer-notification-clone \
then ceilometer-compute-clone require-all=false
pcs constraint order start libvirtd-compute-clone then \
ceilometer-compute-clone
pcs constraint colocation add ceilometer-compute-clone with \

```

```

libvirt-d-compute-clone
//创建NFS文件系统资源, 用以nova-compute的共享存储
pcs resource create nova-compute-fs Filesystem \
device="$master:$nfs_dir/instances" \
directory="/var/lib/nova/instances" fstype="nfs" options="v3" op start \
timeout=240 --clone interleave=true --disabled --force
pcs constraint location nova-compute-fs-clone rule \
resource-discovery=exclusive score=0 node_role eq compute
pcs constraint order start ceilometer-compute-clone then \
nova-compute-fs-clone
pcs constraint colocation add nova-compute-fs-clone with \
ceilometer-compute-clone
//创建nova-compute资源, openstack-nova-compute服务由NovaCompute替代
pcs resource create nova-compute ocf:openstack:NovaCompute \
auth_url=http://$vip_keystone:35357 username=admin password=admin \
tenant_name=admin op start timeout=300 --clone interleave=true \
--disabled --force
pcs constraint location nova-compute-clone rule \
resource-discovery=exclusive score=0 node_role eq compute
pcs constraint order start nova-conductor-clone then nova-compute-clone \
require-all=false
pcs constraint order start nova-compute-fs-clone then \
nova-compute-clone require-all=false
pcs constraint colocation add nova-compute-clone with \
nova-compute-fs-clone
pcs constraint colocation add nova-compute-clone with \
libvirt-d-compute-clone
pcs constraint order start libvirt-d-compute-clone then \
nova-compute-clone require-all=false
pcs constraint order start nova-compute-clone then nova-evacuate \
require-all=false

```

对于生产环境, 计算节点 Fencing 至关重要(虚拟机实验环境可以略过此步骤, 采取手工方式隔离)。完成计算节点 Fencing 操作的 Stonith 设备创建如下:

```

pcs stonith create fence-computer1 \
fence_ipmilan pcmk_host_list= computer1 ipaddr=$computer1_ip \
login=$IPMILAN_USERNAME passwd=$IPMILAN_PASSWORD lanplus=1 cipher=1 \
op monitor interval=60s

```

创建一个独立的 fence-nova Stonith 设备, 如下:

```

pcs stonith create fence-nova fence_compute \
auth-url=http://$vip_keystone:35357 login=admin passwd=admin \
tenant-name=admin record-only=1 action=off --force

```

确保计算节点被 Fencing 之后, 能够重新恢复到 Pacemaker 集群中。即迫使隔离后的计算节点在恢复后, 重新自动加入集群, 如下:

```

pcs property set cluster-recheck-interval=1min

```

将计算节点加入 Pacemaker 集群，使其成为 Pacemaker 集群的远程节点，并由 Pacemaker 管理控制计算节点上运行的资源。为了确保属于计算层面的 Openstack 资源仅运行在计算节点，需要将 Pacemaker 集群中的远程节点打上“computer”节点属性标签，具体过程参考如下：

```
for node in $computer_nodes_num
do
    //将计算节点加入Pacemaker集群
    pcs resource create $node ocf:pacemaker:remote \
    reconnect_interval=60 op monitor interval=20
    //为计算节点设置“computer”节点属性标签
    pcs property set --node $node node_role=compute
done
```

一切准备就绪之后，启动控制层面和计算层面的全部 Pacemaker 资源，如下：

```
pcs resource enable keystone
pcs resource enable neutron-openvswitch-agent-compute
pcs resource enable libvirt-d-compute
pcs resource enable ceilometer-compute
pcs resource enable nova-compute-fs
pcs resource enable nova-compute
```

待 Pacemaker 集群资源重新启动并运行稳定后，可以通过 Pacemaker 集群命令查看当前 Pacemaker 集群中各个 OpenStack 相关服务的高可用运行情况。如果完全按照本书第 11 章和本章介绍的内容进行 OpenStack 高可用资源的部署创建，则截至目前，Pacemaker 集群的运行状态应该如下所示：

```
[root@controller1-vm openstack]# pcs status
Cluster name: openstack-ha
Last updated: Mon Dec 5 05:29:19 2016      Last change: Mon Dec 5 05:20:48
2016 by hacluster via crmd on controller1-vm
Stack: corosync
Current DC: controller2-vm (version 1.1.13-10.el7-44eb2dd) - partition with quorum
5 nodes and 228 resources configured

Online: [ controller1-vm controller2-vm controller3-vm ]
RemoteOnline: [ computer1 computer2 ]

Full list of resources:
fence1 (stonith:fence_xvm):      Started controller1-vm
fence2 (stonith:fence_xvm):      Started controller2-vm
fence3 (stonith:fence_xvm):      Started controller3-vm
Clone Set: lb-haproxy-clone [lb-haproxy]
    Started: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]
vip-db (ocf::heartbeat:IPaddr2):      Started controller3-vm
vip-rabbitmq (ocf::heartbeat:IPaddr2):      Started controller1-vm
vip-keystone (ocf::heartbeat:IPaddr2):      Started controller2-vm
```



```

vip-glance      (ocf::heartbeat:IPaddr2):      Started controller3-vm
vip-cinder      (ocf::heartbeat:IPaddr2):      Started controller1-vm
vip-swift       (ocf::heartbeat:IPaddr2):      Started controller2-vm
vip-neutron     (ocf::heartbeat:IPaddr2):      Started controller3-vm
vip-nova        (ocf::heartbeat:IPaddr2):      Started controller1-vm
vip-horizon     (ocf::heartbeat:IPaddr2):      Started controller2-vm
vip-heat        (ocf::heartbeat:IPaddr2):      Started controller3-vm
vip-ceilometer  (ocf::heartbeat:IPaddr2):      Started controller1-vm
vip-qpid        (ocf::heartbeat:IPaddr2):      Started controller2-vm

Master/Slave Set: galera-master [galera]
    Masters: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]

Clone Set: memcached-clone [memcached]
    Started: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]

Master/Slave Set: rabbitmq-cluster-master [rabbitmq-cluster]
    Masters: [ controller2-vm ]
    Slaves: [ controller3-vm ]
    Stopped: [ computer1 computer2 controller1-vm ]

Clone Set: mongodb-clone [mongodb]
    Started: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]

Clone Set: keystone-clone [keystone]
    Started: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]

Clone Set: glance-fs-clone [glance-fs]
    Started: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]

Clone Set: glance-registry-clone [glance-registry]
    Started: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]

Clone Set: glance-api-clone [glance-api]
    Started: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]

Clone Set: cinder-api-clone [cinder-api]
    Started: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]

Clone Set: cinder-scheduler-clone [cinder-scheduler]
    Started: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]

cinder-volume   (systemd:openstack-cinder-volume):      Started controller3-vm

Clone Set: neutron-server-api-clone [neutron-server-api]
    Started: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]

Clone Set: neutron-ovs-cleanup-clone [neutron-ovs-cleanup]
    Started: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]

Clone Set: neutron-netns-cleanup-clone [neutron-netns-cleanup]
    Started: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]

Clone Set: neutron-openvswitch-agent-clone [neutron-openvswitch-agent]

```

```

Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: neutron-dhcp-agent-clone [neutron-dhcp-agent]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: neutron-l3-agent-clone [neutron-l3-agent]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: neutron-metadata-agent-clone [neutron-metadata-agent]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: nova-api-clone [nova-api]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: nova-scheduler-clone [nova-scheduler]
Started: [ controller1-vm controller2-vm ]
Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: nova-conductor-clone [nova-conductor]
Started: [ controller1-vm controller2-vm ]
Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: nova-consoleauth-clone [nova-consoleauth]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: nova-novncproxy-clone [nova-novncproxy]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: nova-cert-clone [nova-cert]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Master/Slave Set: redis-server-master [redis-server]
Masters: [ controller3-vm ]
Slaves: [ controller1-vm controller2-vm ]
Stopped: [ computer1 computer2 ]
vip-redis (ocf::heartbeat:IPaddr2): Started controller3-vm
Clone Set: ceilometer-central-clone [ceilometer-central]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: ceilometer-collector-clone [ceilometer-collector]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: ceilometer-api-clone [ceilometer-api]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: ceilometer-delay-clone [ceilometer-delay]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: ceilometer-alarm-evaluator-clone [ceilometer-alarm-evaluator]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: ceilometer-alarm-notifier-clone [ceilometer-alarm-notifier]
Started: [ controller1-vm controller2-vm controller3-vm ]

```

```

Stopped: [ computer1 computer2 ]
Clone Set: ceilometer-notification-clone [ceilometer-notification]
  Started: [ controller1-vm controller2-vm controller3-vm ]
  Stopped: [ computer1 computer2 ]
Clone Set: heat-api-clone [heat-api]
  Started: [ controller1-vm controller2-vm controller3-vm ]
  Stopped: [ computer1 computer2 ]
Clone Set: heat-api-cfn-clone [heat-api-cfn]
  Started: [ controller1-vm controller2-vm controller3-vm ]
  Stopped: [ computer1 computer2 ]
Clone Set: heat-api-cloudwatch-clone [heat-api-cloudwatch]
  Started: [ controller1-vm controller2-vm controller3-vm ]
  Stopped: [ computer1 computer2 ]
Clone Set: heat-engine-clone [heat-engine]
  Started: [ controller1-vm controller2-vm controller3-vm ]
  Stopped: [ computer1 computer2 ]
Clone Set: horizon-clone [horizon]
  Started: [ controller1-vm controller2-vm controller3-vm ]
  Stopped: [ computer1 computer2 ]
nova-evacuate (ocf::openstack:NovaEvacuate): Started controller1-vm
Clone Set: neutron-openvswitch-agent-compute-clone [neutron-openvswitch-agent-compute]
  Started: [ computer1 computer2 ]
  Stopped: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: libvirtd-compute-clone [libvirtd-compute]
  Started: [ computer1 computer2 ]
  Stopped: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: ceilometer-compute-clone [ceilometer-compute]
  Started: [ computer1 computer2 ]
  Stopped: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: nova-compute-fs-clone [nova-compute-fs]
  Started: [ computer1 computer2 ]
  Stopped: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: nova-compute-clone [nova-compute]
  Started: [ computer1 computer2 ]
  Stopped: [ controller1-vm controller2-vm controller3-vm ]
fence-nova      (stonith:fence_compute):      Started controller2-vm
computer1      (ocf::pacemaker:remote):      Started controller1-vm
computer2      (ocf::pacemaker:remote):      Started controller2-vm

```

PCSD Status:

```

controller1-vm: Online
controller2-vm: Online
controller3-vm: Online

```

Daemon Status:

```

corosync: active/enabled
pacemaker: active/enabled
pcsd: active/enabled

```

由上述 Pacemaker 集群运行状态输出可以看到，集群共有五个节点组成，其中三个集

群节点（即三个控制节点 `controller1-vm`、`controller2-vm` 和 `controller3-vm`），两个远程节点（即两个计算节点 `computer1` 和 `computer2`）。此外，集群中共配置了 228 个资源，当前的 DC 节点为 `controller2-vm` 控制节点。而且集群资源被划分为两大类，即控制层面资源和计算层面资源，其中控制层面资源仅运行在三个控制节点 `controller1-vm`、`controller2-vm` 和 `controller3-vm` 上，而计算层面资源仅运行在两个计算节点 `computer1` 和 `computer2` 上。至此，基于 Pacemaker 的 OpenStack 高可用集群已经全部部署完成，各个 OpenStack 所依赖的基础服务，以及 OpenStack 控制节点服务和计算节点服务均以各种高可用模式运行在不同的节点之上，下一节中，我们将对现已部署完成的 OpenStack 高可用集群进行各项云计算功能的正确性和高可用性的验证。另外，本节介绍的 Openstack 计算节点服务高可用部署源代码可参考笔者位于 Github 上的开源项目（<https://github.com/ynwssjx/Openstack-HA-Deployment>），具体部署脚本为 `15_openstack_computer_nodes_ha_main.sh`，执行该脚本后，Openstack 计算节点将会自动部署高可用的 Nova-compute 服务。

12.3 OpenStack 集群服务高可用验证

常规 OpenStack 服务多节点分布式部署模式中，OpenStack 各个相关服务由操作系统直接控制（如 Systemd），而在 OpenStack 集群服务高可用部署模式中，OpenStack 集群服务交由第三方集群管理软件控制（如 Pacemaker）。在本书介绍的 OpenStack 高可用部署模式中，我们采用了基于 Pacemaker 集群管理技术的 OpenStack 高可用部署模式，相对其他高可用方案，基于 Pacemaker 的技术方案更为直观并易于理解实现，同时也具有更多的用户部署经验和开源厂商的支持。而在开源社区中，RedHat 的相关开源软件和 OpenStack 的 RDO 发行版本对 Pacemaker 及其各种高可用相关的 OCF 脚本进行了大力支持，因此采用基于 Pacemaker 集群技术的 OpenStack 高可用部署方案，不仅具有丰富的一线部署案例和经验，同时还具有强大的上游开发团队支持。

本章采用 Pacemaker 集群技术，部署实现了由三个控制节点和两个计算节点组成的 OpenStack 高可用集群，集群服务的高可用性不仅覆盖了 OpenStack 相关的核心服务，同时还实现了 OpenStack 所依赖的数据库和消息队列等基础服务的高可用性，真正实现了由底至上的全栈高可用。在 11.3 节中，OpenStack 相关的全部服务在 Pacemaker 集群中已成功部署完成，并全部以各种高可用模式运行在 Pacemaker 集群中，本节将从 OpenStack 云计算功能的正确性和高可用性两个维度对已部署实现的 OpenStack 高可用集群进行验证。

12.3.1 OpenStack 高可用集群功能性验证

在对 OpenStack 集群的高可用性进行验证之前，必须保证集群所提供的各项云计算功能是正常的。为了对 OpenStack 进行综合性功能验证，这里将以 admin 用户的身份在 OpenStack 命令行和 Dashboard 环境中分别创建镜像、创建 Volume、创建网络对象（网络、

子网和路由等)并最终创建实例,在验证之前,需要保证 Pacemaker 高可用集群各项资源正常运行,验证步骤如下:

1. Pacemaker 高可用集群资源配置及运行状态确认

此步骤主要检查 Pacemaker 集群资源配置、资源约束、集群属性和资源运行状态是否已如预期设置和运行。在本章部署实现的 OpenStack 高可用集群中,各项 Pacemaker 集群信息的查看方式和结果如下。

Pacemaker 集群属性信息确认查看:

```
[root@controller1-vm ~(keystone_admin)]$ pcs property
```

Cluster Properties:

```
cluster-infrastructure: corosync
cluster-name: openstack-ha
cluster-recheck-interval: 1min
dc-version: 1.1.13-10.el7-44eb2dd
have-watchdog: false
last-lrm-refresh: 1481029550
redis-server_REPL_INFO: controller3-vm
stonith-enabled: true
```

Node Attributes:

```
computer1: node_role=compute
computer2: node_role=compute
controller1-vm: node_role=controller
controller2-vm: node_role=controller
controller3-vm: node_role=controller
```

Pacemaker 集群资源约束 (Location、Order 和 Colocation 约束) 结果确认查看:

```
[root@controller1-vm ~(keystone_admin)]$ pcs constraint
```

Location Constraints:

```
Resource: ceilometer-alarm-evaluator-clone
Constraint: location-ceilometer-alarm-evaluator-clone (resource-discovery=
exclusive)
Rule: score=0
Expression: node_role eq controller
Resource: ceilometer-alarm-notifier-clone
Constraint: location-ceilometer-alarm-notifier-clone (resource-discovery=
exclusive)
Rule: score=0
Expression: node_role eq controller
Resource: ceilometer-api-clone
```

.....

Ordering Constraints:

```
start vip-db then start lb-haproxy-clone (kind:Optional)
start vip-rabbitmq then start lb-haproxy-clone (kind:Optional)
start vip-keystone then start lb-haproxy-clone (kind:Optional)
start vip-glance then start lb-haproxy-clone (kind:Optional)
start vip-cinder then start lb-haproxy-clone (kind:Optional)
start vip-swift then start lb-haproxy-clone (kind:Optional)
start vip-neutron then start lb-haproxy-clone (kind:Optional)
```

```

start vip-nova then start lb-haproxy-clone (kind:Optional)
start vip-horizon then start lb-haproxy-clone (kind:Optional)
start vip-heat then start lb-haproxy-clone (kind:Optional)
start vip-ceilometer then start lb-haproxy-clone (kind:Optional)
start vip-qpid then start lb-haproxy-clone (kind:Optional)
start lb-haproxy-clone then start galera-master (kind:Mandatory)
start glance-fs-clone then start glance-registry-clone (kind:Mandatory)
start glance-registry-clone then start glance-api-clone (kind:Mandatory)
start cinder-api-clone then start cinder-scheduler-clone (kind:Mandatory)
start cinder-scheduler-clone then start cinder-volume (kind:Mandatory)

```

.....

Colocation Constraints:

```

vip-db with lb-haproxy-clone (score:INFINITY)
vip-rabbitmq with lb-haproxy-clone (score:INFINITY)
vip-keystone with lb-haproxy-clone (score:INFINITY)
vip-glance with lb-haproxy-clone (score:INFINITY)
vip-cinder with lb-haproxy-clone (score:INFINITY)
vip-swift with lb-haproxy-clone (score:INFINITY)
vip-neutron with lb-haproxy-clone (score:INFINITY)
vip-nova with lb-haproxy-clone (score:INFINITY)
vip-horizon with lb-haproxy-clone (score:INFINITY)
vip-heat with lb-haproxy-clone (score:INFINITY)
vip-ceilometer with lb-haproxy-clone (score:INFINITY)
vip-qpid with lb-haproxy-clone (score:INFINITY)
glance-registry-clone with glance-fs-clone (score:INFINITY)
glance-api-clone with glance-registry-clone (score:INFINITY)
cinder-scheduler-clone with cinder-api-clone (score:INFINITY)
cinder-volume with cinder-scheduler-clone (score:INFINITY)
neutron-netns-cleanup-clone with neutron-ovs-cleanup-clone (score:INFINITY)
neutron-openvswitch-agent-clone with neutron-netns-cleanup-clone (score:
INFINITY)
neutron-dhcp-agent-clone with neutron-openvswitch-agent-clone (score:
INFINITY)
neutron-l3-agent-clone with neutron-dhcp-agent-clone (score:INFINITY)
neutron-metadata-agent-clone with neutron-l3-agent-clone (score:INFINITY)

```

.....

完整的 Pacemaker 集群约束输出结果可参考笔者上传至 Github (<https://github.com/ynwssjx/Openstack-HA-Deployment>) 的集群约束输出结果文件 (文件名称为 constraints_final)。

Pacemaker 集群资源及其运行情况查看确认:

```

[root@controller1-vm ~(keystone_admin)]$ pcs resource
Clone Set: lb-haproxy-clone [lb-haproxy]
  Started: [ controller1-vm controller2-vm controller3-vm ]
  Stopped: [ computer1 computer2 ]
vip-db (ocf::heartbeat:IPaddr2): Started controller3-vm
vip-rabbitmq (ocf::heartbeat:IPaddr2): Started controller1-vm
vip-keystone (ocf::heartbeat:IPaddr2): Started controller2-vm
vip-glance (ocf::heartbeat:IPaddr2): Started controller3-vm
vip-cinder (ocf::heartbeat:IPaddr2): Started controller1-vm
vip-swift (ocf::heartbeat:IPaddr2): Started controller2-vm

```



```

vip-neutron      (ocf::heartbeat:IPAddr2):      Started controller3-vm
vip-nova         (ocf::heartbeat:IPAddr2):      Started controller1-vm
vip-horizon      (ocf::heartbeat:IPAddr2):      Started controller2-vm
vip-heat         (ocf::heartbeat:IPAddr2):      Started controller3-vm
vip-ceilometer   (ocf::heartbeat:IPAddr2):      Started controller1-vm
vip-qpid         (ocf::heartbeat:IPAddr2):      Started controller2-vm
Master/Slave Set: galera-master [galera]
    Masters: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]
Clone Set: memcached-clone [memcached]
    Started: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]
Master/Slave Set: rabbitmq-cluster-master [rabbitmq-cluster]
    Masters: [ controller2-vm ]
    Slaves: [ controller1-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]
Clone Set: mongodb-clone [mongodb]
    Started: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]
Clone Set: keystone-clone [keystone]
    Started: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]
Clone Set: glance-fs-clone [glance-fs]
    Started: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]
Clone Set: glance-registry-clone [glance-registry]
    Started: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]
Clone Set: glance-api-clone [glance-api]
    Started: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]
Clone Set: cinder-api-clone [cinder-api]
    Started: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]
Clone Set: cinder-scheduler-clone [cinder-scheduler]
    Started: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]
cinder-volume    (systemd:openstack-cinder-volume):      Started controller3-vm
Clone Set: neutron-server-api-clone [neutron-server-api]
    Started: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]
Clone Set: neutron-ovs-cleanup-clone [neutron-ovs-cleanup]
    Started: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]
Clone Set: neutron-netns-cleanup-clone [neutron-netns-cleanup]
    Started: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]
Clone Set: neutron-openvswitch-agent-clone [neutron-openvswitch-agent]
    Started: [ controller1-vm controller2-vm controller3-vm ]
    Stopped: [ computer1 computer2 ]
Clone Set: neutron-dhcp-agent-clone [neutron-dhcp-agent]

```

```

Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: neutron-l3-agent-clone [neutron-l3-agent]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: neutron-metadata-agent-clone [neutron-metadata-agent]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: nova-api-clone [nova-api]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: nova-scheduler-clone [nova-scheduler]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: nova-conductor-clone [nova-conductor]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: nova-consoleauth-clone [nova-consoleauth]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: nova-novncproxy-clone [nova-novncproxy]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: nova-cert-clone [nova-cert]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Master/Slave Set: redis-server-master [redis-server]
Masters: [ controller3-vm ]
Slaves: [ controller1-vm controller2-vm ]
Stopped: [ computer1 computer2 ]
vip-redis (ocf::heartbeat:IPaddr2): Started controller3-vm
Clone Set: ceilometer-central-clone [ceilometer-central]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: ceilometer-collector-clone [ceilometer-collector]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: ceilometer-api-clone [ceilometer-api]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: ceilometer-delay-clone [ceilometer-delay]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: ceilometer-alarm-evaluator-clone [ceilometer-alarm-evaluator]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: ceilometer-alarm-notifier-clone [ceilometer-alarm-notifier]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: ceilometer-notification-clone [ceilometer-notification]

```

```

Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: heat-api-clone [heat-api]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: heat-api-cfn-clone [heat-api-cfn]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: heat-api-cloudwatch-clone [heat-api-cloudwatch]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: heat-engine-clone [heat-engine]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Clone Set: horizon-clone [horizon]
Started: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
nova-evacuate (ocf::openstack:NovaEvacuate): Started controller1-vm
Clone Set: neutron-openvswitch-agent-compute-clone [neutron-openvswitch-agent-compute]
Started: [ computer1 computer2 ]
Stopped: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: libvirt-compute-clone [libvirt-compute]
Started: [ computer1 computer2 ]
Stopped: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: ceilometer-compute-clone [ceilometer-compute]
Started: [ computer1 computer2 ]
Stopped: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: nova-compute-fs-clone [nova-compute-fs]
Started: [ computer1 computer2 ]
Stopped: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: nova-compute-clone [nova-compute]
Started: [ computer1 computer2 ]
Stopped: [ controller1-vm controller2-vm controller3-vm ]
computer1 (ocf::pacemaker:remote): Started controller1-vm
computer2 (ocf::pacemaker:remote): Started controller2-vm

```

在一个基于 Pacemaker 的高可用 OpenStack 集群中，各项服务正常启动和运行时，Pacemaker 集群中的资源运行情况应该如上所示。管理员需要确认每一项 Pacemaker 资源都正常启动，并运行在正确的节点上（此处重点确认资源是否完全启动以及是否由正确的节点启动）。由于 Pacemaker 将节点和资源划分为控制层面和计算层面，因此务必确认控制层面的资源仅运行在控制层面节点上，而计算层面的资源仅运行在计算节点上。

2. OpenStack 服务启动情况确认

在高可用环境下，OpenStack 服务完全交由 Pacemaker 控制，而不再是 Systemd 等系统进程控制。在 Pacemaker 资源正常运行后，可以通过 OpenStack 各个服务项目的命令行客户端再次确认相关服务的运行情况。首先查看 Nova 服务运行情况，重点检查 Nova-compute 服务是否在各个计算节点正常启动，Nova-api 等控制层面的服务是否在三个控制节

点正常启动，查看方式如下：

```
[root@controller1-vm ~(keystone_admin)]$ nova service-list
```

Id	Binary	Host	Zone	Status	State
1	nova-consoleauth	controller3-vm	internal	enabled	up
3	nova-consoleauth	controller1-vm	internal	enabled	up
5	nova-consoleauth	controller2-vm	internal	enabled	up
6	nova-cert	controller1-vm	internal	enabled	up
8	nova-scheduler	controller1-vm	internal	enabled	up
11	nova-cert	controller3-vm	internal	enabled	up
14	nova-scheduler	controller3-vm	internal	enabled	up
16	nova-conductor	controller1-vm	internal	enabled	up
22	nova-conductor	controller3-vm	internal	enabled	up
25	nova-scheduler	controller2-vm	internal	enabled	up
26	nova-cert	controller2-vm	internal	enabled	up
27	nova-conductor	controller2-vm	internal	enabled	up
30	nova-compute	computer2	nova	enabled	up
33	nova-compute	computer1	nova	enabled	up

其次，检查 Neutron 网络服务 Agents 运行情况，重点查看 L3、DHCP 和 Metadata agent 是否在三个控制节点上正常启动，而 Open vSwitch agent 是否在全部节点上正常启动，查看方式如下：

```
[root@controller1-vm ~(keystone_admin)]$ neutron agent-list
```

id	agent_type	host	alive	binary
...d62	DHCP agent	controller1-vm	:-)	neutron-dhcp-agent
...d39	Open vSwitch agent	controller2-vm	:-)	neutron-openvswitch-agent
...90d	Metadata agent	controller1-vm	:-)	neutron-metadata-agent
...7eb	Open vSwitch agent	controller1-vm	:-)	neutron-openvswitch-agent
...ac7	L3 agent	controller2-vm	:-)	neutron-l3-agent
...f0d	Metadata agent	controller2-vm	:-)	neutron-metadata-agent
...3a5	Open vSwitch agent	computer2	:-)	neutron-openvswitch-agent
...79b	Metadata agent	controller3-vm	:-)	neutron-metadata-agent
...58a	DHCP agent	controller2-vm	:-)	neutron-dhcp-agent
...1e5	Open vSwitch agent	controller3-vm	:-)	neutron-openvswitch-agent
...267	DHCP agent	controller3-vm	:-)	neutron-dhcp-agent
...000	L3 agent	controller3-vm	:-)	neutron-l3-agent
...3fd	L3 agent	controller1-vm	:-)	neutron-l3-agent
...3b1	Open vSwitch agent	computer1	:-)	neutron-openvswitch-agent

最后，检查 Cinder 块存储服务运行情况，重点查看 Scheduler 服务是否正常启动，各个 Volume 后端驱动服务是否正常启动（Cinder-volume 服务以 A/P 模式运行仅运行在一个控制节点上），本章仅部署了 NFS 一种后端驱动，查看方式如下：

```
[root@controller1-vm ~(keystone_admin)]$ cinder service-list
```

Binary	Host	Zone	Status	State	Updated_at
cinder-scheduler	openstack-cinder	nova	enabled	up	2016-12-06T1...
cinder-volume	openstack-cinder	nova	enabled	up	2016-12-06T1...

3. 创建 Glance 镜像

通过上传镜像至 Glance 中, 验证高可用部署模式下 OpenStack 镜像服务 Glance (Glance-api 和 Glance-registry 服务) 是否正常工作, 这里以测试镜像 Cirros 为例来验证 Glance 服务, 镜像创建过程如下:

```
[root@controller1-vm ~(keystone_admin)]$ glance image-create \
--container-format bare --disk-format qcow2 --is-public true --file \
cirros-0.3.4-x86_64-disk.img --name cirros --progress
```

镜像上传完成后, 查询并检索 OpenStack 镜像服务 Glance 中存储的镜像, 查询结果如下:

```
[root@controller1-vm ~(keystone_admin)]$ glance image-list
```

ID	Name	Disk Format	Container Format	Size	Status
...e9c018	cirros	qcow2	bare	13287936	active

4. 创建 Cinder 块存储

通过块存储创建, 验证高可用部署模式下 OpenStack 块存储服务 Cinder 是否正常工作。这里需要重点强调的是, Cinder 项目仅有 Cinder-api 服务以 A/A 模式运行并实现在三个控制节点之间的负载均衡, 而 Cinder-volume 服务仅以 A/P 模式运行在一个控制节点上。此处创建大小为 2GB, 名称为 NFS-Volume1 的 Volume 块存储, 创建过程如下:

```
[root@controller1-vm ~(keystone_admin)]$ cinder create --display-name NFS-Volume1 2
[root@controller1-vm ~(keystone_admin)]$ cinder list
```

ID	Status	Display Name	Size	Volume Type	Bootable	Attached to
...9cd2	available	NFS-Volume1	2	-	false	

5. 创建 Neutron 网络

通过网络创建, 验证高可用部署模式下 OpenStack 网络服务 Neutron 是否正常工作。网络创建包括外网及其子网创建、内网及其子网创建、高可用 L3 Router 创建以及内外网与 Router 的关联。首先创建一个 Flat 类型的外部网络, 创建过程如下:

```
[root@controller1-vm ~(keystone_admin)]$ neutron net-create ext-net \
--router:external --provider:physical_network external \
```

```
--provider:network_type flat
```

在外部网络 `ext-net` (192.168.115.0/24) 中创建一个外网子网, 网关为 192.168.115.254, 子网 IP 地址段为 192.168.115.200~192.168.115.200, 外网子网创建过程如下:

```
[root@controller1-vm ~(keystone_admin)]$ neutron subnet-create ext-net \
192.168.115.0/24 --name ext-subnet --allocation-pool \
start=192.168.115.200,end=192.168.115.250 --disable-dhcp --gateway \
192.168.115.254
```

外网是基于外部物理网络所创建的虚拟网络 (与外部物理网络可以相互通信), 这里再创建一个租户自定义的内部网络, 内网完全由租户自定义, 而且不会影响到实例与外界通信, 租户内网创建过程如下:

```
[root@controller1-vm ~(keystone_admin)]$ neutron net-create admin-net
```

同 `ext-net` 外网一样, `admin-net` 内网也需要创建一个子网, 并在子网中定义此网络可用的 IP 地址范围等信息, 内网子网创建过程如下:

```
[root@controller1-vm ~(keystone_admin)]$ neutron subnet-create \
admin-net 192.128.1.0/24 --name admin-subnet --gateway 192.128.1.1
```

租户内部网络中的虚拟机要与外界互相通信, 则必须存在 L3 Router。在本文介绍的 Neutron 网络高可用部署模式中, 采用 L3 HA 高可用模式实现 L3 Agent 的分布式高可用, 因此, 这里创建的路由将具备高可用性, 高可用路由创建过程如下:

```
[root@controller1-vm ~(keystone_admin)]$ neutron router-create admin-router
```

网络和路由均创建完成后, 需要为 L3 路由设置外网网关和内网接口, 以便内网中的实例可以通过 L3 路由与外网进行 L3 通信, 路由 `admin-router` 的网关及接口设置过程如下:

```
[root@controller1-vm ~(keystone_admin)]$ neutron router-interface-add \
admin-router admin-subnet
[root@controller1-vm ~(keystone_admin)]$ neutron router-gateway-set \
admin-router ext-net
```

现在, Neutron 基本网络已经创建并设置完毕, 即针对特定租户 (这里为 `admin` 租户) 的 OpenStack 网络环境已经准备就绪。由于采用的是网络高可用部署模式, 租户在 Neutron 中创建网络时, Neutron 会自动创建一个 HA 网络, 即除了云管理员创建的外网和租户创建的内网之外, Neutron 还自动创建了一个 HA 网络, 如下:

```
[root@controller1-vm ~(keystone_admin)]$ neutron net-list
```

id	name	subnets
...773	admin-net	...-0e4c14e4f1d5 192.128.1.0/24
...3ed	ext-net	...-3f8e3b4cd763 192.168.115.0/24
...b3a	HA network tenant ...5135	...-19f437530383 169.254.192.0/18

由于在配置文件中启用了 `L3_HA=True` 配置项，因此创建的 Router 应该具有 `HA=True` 的属性（`Distributed` 是 DVR 高可用模式相关的属性，这里为 `False`），如下：

```
[root@controller1-vm ~(keystone_admin)]$ neutron router-list
```

id	name	external_gateway_info	distributed	ha
...884	admin-router	False	True

在 L3 HA 高可用模式下，L3 Agent 将同时运行在多个网络控制节点上（这里网络控制节点即 OpenStack 集群的三个控制节点），租户创建的每个路由都会以相同的配置信息分布在各个网络控制节点上，但是对于每一个租户路由，正常情况下仅有一个 L3 Agent 提供 L3 服务，其他 L3 Agent 均处于 Standby 状态，如下：

```
[root@controller1-vm ~(keystone_admin)]$ neutron \
l3-agent-list-hosting-router admin-router
```

id	host	admin_state_up	alive	ha_state
...a95bda3fd	controller1-vm	True	:-)	standby
...203bb8000	controller3-vm	True	:-)	standby
...e64f20ac7	controller2-vm	True	:-)	active

由于 L3 HA 高可用模式会将每个 Router 分布到各个控制节点，因此在每个控制节点的网络命名空间中都会有 ID 完全相同的 `qrouter`，此外由于在配置文件中为每个网络设置了三个 DHCP agent，因此每个控制节点中还会有 ID 完全相同的 `qdhcp`，如下：

```
[root@controller1-vm ~]$ ip netns
qrouter-93a07bba-ab77-4253-8a2e-a31332161884
qdhcp-4102d57d-4ab0-4f30-bbb4-5e002c29a773
[root@controller2-vm ~]# ip netns
qrouter-93a07bba-ab77-4253-8a2e-a31332161884
qdhcp-4102d57d-4ab0-4f30-bbb4-5e002c29a773
[root@controller3-vm ~]# ip netns
qrouter-93a07bba-ab77-4253-8a2e-a31332161884
qdhcp-4102d57d-4ab0-4f30-bbb4-5e002c29a773
```

6. 创建 Nova 实例

在镜像、块存储和网络均创建完成后，现在开始创建 Nova 实例虚拟机。实例创建过程是对 OpenStack 各个项目功能是否正常的完整验证，只有当 Keystone、Glance 和 Neutron 等项目均正常运行的前提下，Nova 实例创建才会成功。这里创建名为 `instance1` 的实例，如下：

```
[root@controller1-vm ~(keystone_admin)]$ nova boot --flavor 1 --image\
cirros --nic net-id=4102d57d-4ab0-4f30-bbb4-5e002c29a773 \
```

```
--security-group default --key-name admin-key instance1
```

Nova-api 返回实例创建信息后，并不意味着实例已经创建成功，仅是表明 API 已经将实例创建请求转发至 Scheduler。Scheduler 将根据一定的调度算法选取最佳的计算节点来创建指定的虚拟机，实例在创建过程中会经历多个任务状态，直至最终变为 Active 状态，如下：

```
[root@controller1-vm ~(keystone_admin)]$ nova list
+-----+-----+-----+-----+-----+-----+
| ID | Name | Status | Task State | Power State | Networks |
+-----+-----+-----+-----+-----+-----+
| ...51d | instance1 | BUILD | spawning | NOSTATE | admin-net=192.128.1.5 |
+-----+-----+-----+-----+-----+-----+

[root@controller1-vm ~(keystone_admin)]$ nova list
+-----+-----+-----+-----+-----+-----+
| ID | Name | Status | Task State | Power State | Networks |
+-----+-----+-----+-----+-----+-----+
| ...51d | instance1 | ACTIVE | - | Running | admin-net=192.128.1.5 |
+-----+-----+-----+-----+-----+-----+
```

实例成功创建完成后，可以验证 Cinder 与 Nova 之间的交互是否正常。通常，Nova 实例的持久性存储由 Cinder 提供的 Volume 来实现，实例要往 Volume 写入数据，首先必须将 Volume 挂载到实例，如下过程将 Cinder 块存储 NFS-Volume1 挂载到实例 instance1：

```
[root@controller1-vm ~(keystone_admin)]$ nova volume-attach instance1 0d318bb9-7067-4749-9a96-60318b9f9cd2 auto
+-----+-----+
| Property | Value |
+-----+-----+
| device | /dev/vdb |
| id | 0d318bb9-7067-4749-9a96-60318b9f9cd2 |
| serverId | a6ef2f25-39a9-49c0-ab16-36db95c2451d |
| volumeId | 0d318bb9-7067-4749-9a96-60318b9f9cd2 |
+-----+-----+

[root@controller1-vm ~(keystone_admin)]$ cinder list
+-----+-----+-----+-----+-----+-----+
| ID | Status | Display Name | Size | Bootable | Attached to |
+-----+-----+-----+-----+-----+-----+
| ...cd2 | attaching | NFS-Volume1 | 2 | false | |
+-----+-----+-----+-----+-----+-----+

[root@controller1-vm ~(keystone_admin)]$ cinder list
+-----+-----+-----+-----+-----+-----+
| ID | Status | Display Name | Size | Bootable | Attached to |
+-----+-----+-----+-----+-----+-----+
| ...cd2 | in-use | NFS-Volume1 | 2 | false | ...-36db95c2451d |
+-----+-----+-----+-----+-----+-----+
```

实例 instance1 创建完成后，其网络拓扑如图 12-11 所示。实例在创建完成后便已经接入内网 admin-net 中，而 admin-net 与 L3 路由 admin-router 连接，同时 admin-router 的外

网网关即是 ext-net 网络。因此，instance1 创建之后，实例便可通过 admin-router 网关以 SNAT 形式同外网通信，但是外网却无法与租户内网中的实例 instance1 通信，要实现外网对内网实例的 DNAT 通信，必须为实例创建并绑定一个外网 IP 地址，这个外网 IP 通常称为 Floating IP，即浮动 IP 地址。

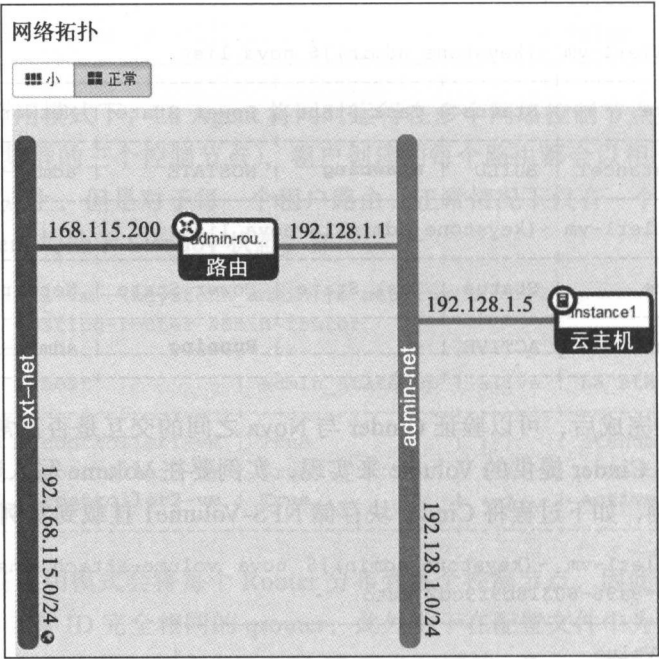


图 12-11 实例 instance1 网络拓扑

浮动 IP 需要在 Neutron 中创建，在 Kilo 版本之前，浮动 IP 由 Neutron 从外网子网中按照一定顺序自动分配，并且用户不能指定。而在 Kilo 以后的版本中，Neutron 支持用户自定义浮动 IP（在子网允许访问内），这一点在实际中非常有用，因为很多应用系统可能已经指定了对外服务的 IP 地址，在 Kilo 版本中，管理员可以在创建浮动 IP 时指定此 IP 并将其关联到虚拟机服务器上（创建 Floating IP 时通过 --floating-ip-address 参数指定）。Neutron 中浮动 IP 自动创建和 Nova 中实例的浮点 IP 关联过程如下：

```
[root@controller1-vm ~(keystone_admin)]$ neutron floatingip-create \
ext-net
Created a new floatingip:
+-----+-----+
| Field          | Value                                     |
+-----+-----+
| fixed_ip_address |                                           |
| floating_ip_address | 192.168.115.201                         |
| floating_network_id | 6eef909a-b843-4653-9d4d-38a269a923ed |
| id              | d1391b67-eba1-4819-b0b0-b32675c0ccb5 |
```

```
| port_id | | |
| router_id | | |
| status | DOWN | |
| tenant_id | 224ef153cdb44985b2634dbd0a3d5135 | |
+-----+-----+-----+
[root@controller1-vm ~(keystone_admin)]$ nova floating-ip-associate
instance1 192.168.115.201
[root@controller1-vm ~(keystone_admin)]$ nova list
+-----+-----+-----+-----+-----+-----+
| ID | Name | Status | Power State | Networks |
+-----+-----+-----+-----+-----+-----+
| ...51d | instance1 | ACTIVE | Running | admin-net=192.128.1.5, 192.168.115.201 |
+-----+-----+-----+-----+-----+-----+
```

以上操作除了可以由管理员在命令行客户端中操作之外，还可以由租户在 OpenStack 的 Dashboard 图形界面中操作（通常与数据中心物理网络相关的外网只能由管理员创建）。这里通过 Dashboard 提供的图形界面创建另外一个实例 instance2，并为其创建和关联浮动 IP。其中，实例 instance2 的图形界面创建过程如图 12-12 所示，用户只需简单地选择可用域（默认为 Nova）、实例名称、主机类型、创建的主机数量、主机启动源等参数，如果是镜像启动则选择镜像名称，然后选择已经创建好的租户网络和安全策略组即可创建实例。

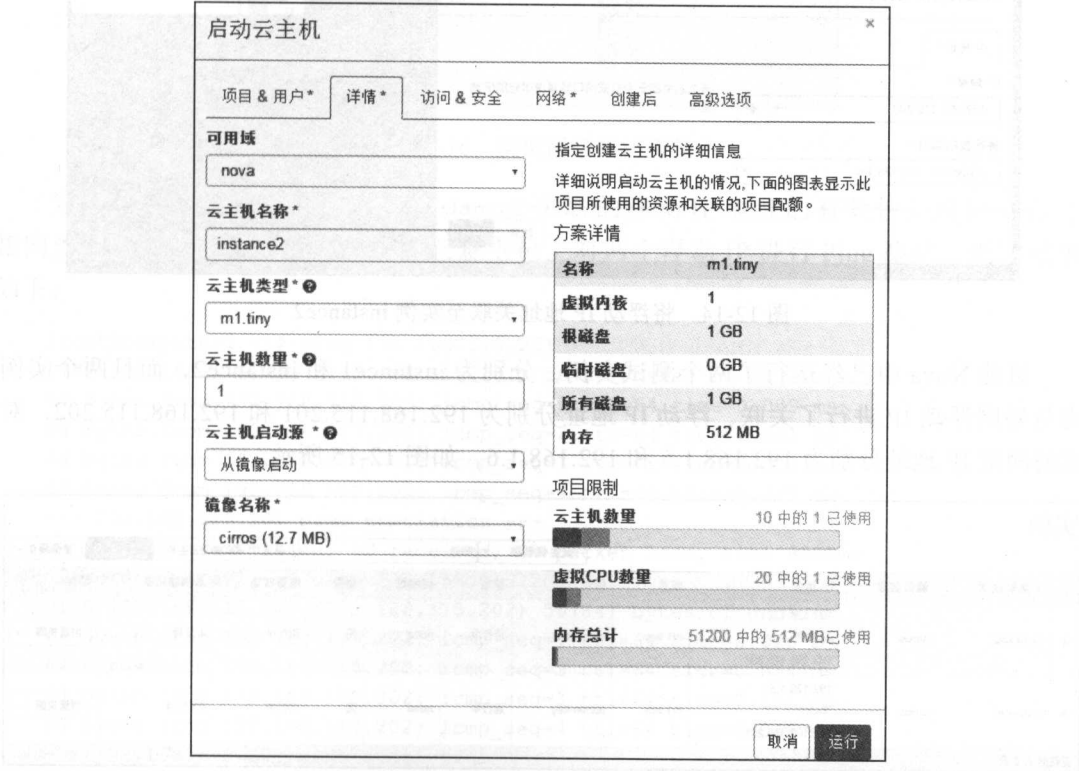


图 12-12 在 Dashboard 上创建实例 instance2

实例 instance2 创建完成后，同样在 Dashboard 上为其创建一个浮动 IP，如图 12-13 所示，创建过程非常简单，只需选择要生成浮动 IP 的一个外部网络，点击“分配 IP”，OpenStack 将自动为用户从指定的外网中创建一个浮动 IP 地址，创建成功之后在右上角会有创建成功的提示，并可以看到分配给用户的具体浮动 IP 地址，如图 12-14 所示，从“待连接的端口”中找到要关联浮动 IP 的实例端口，点击“关联”，OpenStack 将自动为实例关联浮动 IP。

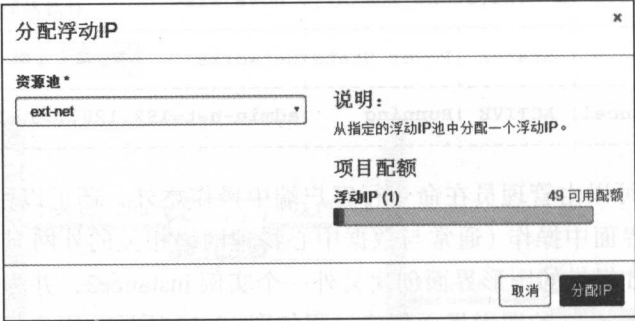


图 12-13 在 Dashboard 中创建浮动 IP

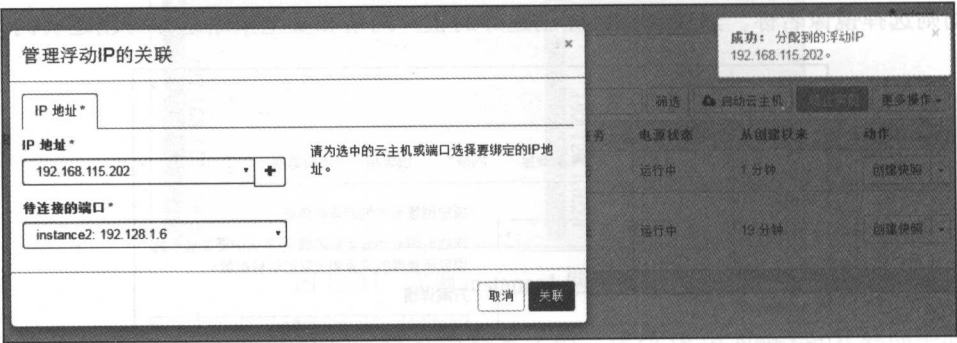


图 12-14 将浮动 IP 地址关联至实例 instance2

目前 Nova 中已经运行了两个测试实例，分别为 instance1 和 instance2，而且两个实例均与外网浮动 IP 进行了关联，浮动 IP 地址分别为 192.168.115.201 和 192.168.115.202，对应的固定 IP 地址分别为 192.168.1.5 和 192.168.1.6，如图 12-15 所示。

实例

云主机名称 筛选 筛选 启动云主机 禁止实例 更多操作										
云主机名称	镜像名称	IP 地址	配置	密钥	状态	可用域	任务	电源状态	从创建以来	动作
instance2	ciros	192.128.1.6 浮动IP: 192.168.115.202	m1.tiny	admin-key	运行中	nova	无	运行中	4 分钟	创建快照
instance1	ciros	192.128.1.5 浮动IP: 192.168.115.201	m1.tiny	admin-key	运行中	nova	无	运行中	23 分钟	创建快照

正在显示 2 项

图 12-15 Dashboard 中的实例运行情况

在本节创建的两个实例中，实例 instance1 和 instance2 均属于同一个租户网络，因此两个实例与外界的相互通信都由 L3 路由 admin-router 实现，两个实例的 Neutron 网络拓扑如图 12-16 所示。

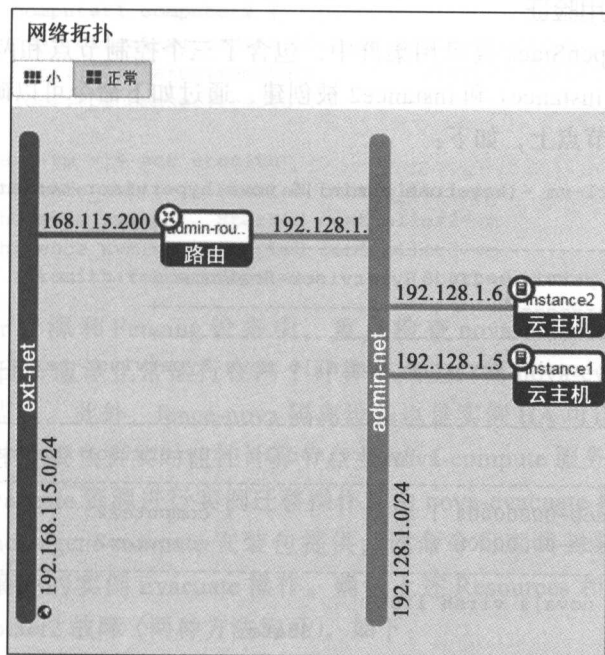


图 12-16 实例网络拓扑情况

为了验证与实例 instance1 与 instance2 关联的浮动 IP 地址的有效性，可以从位于相同外网（192.168.115.0/24）中的任意主机上对两个浮动 IP 进行 Ping 测试，测试结果如下：

```
[root@computer1 ~]# ping 192.168.115.201
PING 192.168.115.201 (192.168.115.201) 56(84) bytes of data.
64 bytes from 192.168.115.201: icmp_seq=1 ttl=64 time=0.029 ms
64 bytes from 192.168.115.201: icmp_seq=2 ttl=64 time=0.041 ms
64 bytes from 192.168.115.201: icmp_seq=3 ttl=64 time=0.053 ms
64 bytes from 192.168.115.201: icmp_seq=4 ttl=64 time=0.029 ms
--- 192.168.115.201 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9008ms
[root@computer1 ~]# ping 192.168.115.202
PING 192.168.115.202 (192.168.115.202) 56(84) bytes of data.
64 bytes from 192.168.115.202: icmp_seq=1 ttl=64 time=0.026 ms
64 bytes from 192.168.115.202: icmp_seq=2 ttl=64 time=0.054 ms
64 bytes from 192.168.115.202: icmp_seq=3 ttl=64 time=0.151 ms
64 bytes from 192.168.115.202: icmp_seq=4 ttl=64 time=0.046 ms
--- 192.168.115.202 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6008ms
```


12.3.2 OpenStack 高可用集群高可用验证

下面，我们主要从两方面介绍高可用验证。

1. 计算节点高可用验证

在本章部署的 OpenStack 高可用集群中，包含了三个控制节点和两个计算节点，而在功能性验证中，实例 instance1 和 instance2 被创建，通过如下命令可以确认两个实例的宿主机分别位于哪个计算节点上，如下：

```
[root@controller1-vm ~(keystone_admin)]$ nova hypervisor-servers \
computer1
+-----+-----+-----+-----+
| ID | Name | Hypervisor ID | Hypervisor Hostname |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
[root@controller1-vm ~(keystone_admin)]$ nova hypervisor-servers \
computer2
+-----+-----+-----+-----+
| ID | Name | Hypervisor ID | Hypervisor Hostname |
+-----+-----+-----+-----+
|...51d | instance-00000008 | 3 | computer2 |
|...acc | instance-0000000b | 3 | computer2 |
+-----+-----+-----+-----+

[root@computer2 nova]# virsh list

```

Id	Name	State
2	instance-00000008	running
3	instance-0000000b	running

可以看到，实例 instance1 和 instance2 均位于计算节点 computer2 上，而计算节点 computer1 目前没有运行任何实例。假设计算节点 computer2 故障，根据前面 OpenStack 计算节点的高可用设计和部署实现，故障节点 computer2 上的实例应该自动迁移至正常主机 computer1 上，即实例 HA 功能会自动保护故障主机上的实例，现在对其进行验证。在功能验证之前，先检查与计算节点相关的 Pacemaker 资源是否运行正常，如下：

```
[root@controller1-vm ~]# pcs resource
.....
nova-evacuate (ocf::openstack:NovaEvacuate): Started controller2-vm
Clone Set: neutron-openvswitch-agent-compute-clone [neutron-openvswitch-agent-
compute]
Started: [ computer1 computer2 ]
Stopped: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: libvirtd-compute-clone [libvirtd-compute]
Started: [ computer1 computer2 ]
Stopped: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: ceilometer-compute-clone [ceilometer-compute]
Started: [ computer1 computer2 ]
Stopped: [ controller1-vm controller2-vm controller3-vm ]
```

```

Clone Set: nova-compute-fs-clone [nova-compute-fs]
  Started: [ computer1 computer2 ]
  Stopped: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: nova-compute-clone [nova-compute]
  Started: [ computer1 computer2 ]
  Stopped: [ controller1-vm controller2-vm controller3-vm ]
computer1 (ocf::pacemaker:remote): Started controller1-vm
computer2 (ocf::pacemaker:remote): Started controller3-vm

[root@controller1-vm ~]# pcs stonith
fence1 (stonith:fence_xvm): Started controller1-vm
fence2 (stonith:fence_xvm): Started controller2-vm
fence3 (stonith:fence_xvm): Started controller3-vm
fence-nova (stonith:fence_compute): Started controller1-vm

```

上述 Pacemaker 资源和 Fencing 设备中，重点检查 nova-compute 和 nova-evacuate 资源，其中 nova-compute 应该正常运行在两个计算节点上，而 nova-evacuate 运行在控制层面上的任一控制节点上。此外，fence-nova 隔离设备也是实例 HA 可以实现的关键，fence-nova 在实例高可用中主要负责实时监控计算节点上 nova-compute 服务是否故障，如果出现故障则通知 nova-evacuate 资源进行实例迁移操作，由 nova-evacuate 再调用 fence_compute 命令行工具（由 fence-agent-compute 安装包提供，该命令行工具封装了对 Nova Evacuate API 的调用）进行最终的实例 Evacuate 操作。确认上述 Resources 和 Stonith 正常运行后，模拟计算节点 Computer2 故障（两种方法均可），如下：

```

//禁用computer2节点CPU
[root@computer2 ~]# echo c > /proc/sysrq-trigger
//或者直接关闭computer2节点
[root@computer2 ~]# shutdown -h now

```

稍等片刻后，两个实例应该在正常的计算节点（这里为 computer1）上重新启动，进入计算节点 Computer1，通过 Libvirt API 可以查看其上是否已有实例运行，如下：

```

[root@computer1 nova]# virsh list

```

Id	Name	State
2	instance-000000008	running
3	instance-00000000b	running

此外，通过 Nova 提供的 Hypervisor 查询 API 也可以分别查看 computer1 和 computer2 两个 Hypervisor 上是否有实例运行，如下：

```

[root@controller1-vm ~(keystone_admin)]$ nova hypervisor-servers \
computer2
+-----+-----+-----+-----+
| ID | Name | Hypervisor ID | Hypervisor Hostname |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
[root@controller1-vm ~(keystone_admin)]$ nova hypervisor-servers \

```

```
computer1
+-----+-----+-----+-----+
| ID | Name | Hypervisor ID | Hypervisor Hostname |
+-----+-----+-----+-----+
|...51d | instance-00000008 | 3 | computer2 |
|...acc | instance-0000000b | 3 | computer2 |
+-----+-----+-----+-----+
```

可以看到，当计算节点 **computer2** 故障后，之前位于其上的两个实例 **instance1** 和 **instance2** 自动迁移到正常运行的 **computer1** 计算节点上。迁移过程中，两个实例将会出现短暂暂停，直至在新计算节点上重新启动。

2. 控制节点高可用验证

在基于 Pacemaker 的 OpenStack 高可用集群中，控制节点不仅承载除了 Nova 计算服务以外的全部 OpenStack 服务组件^①，还作为运行 Pacemaker 和 Corosync 全栈集群服务的 Pacemaker 集群节点，并且 OpenStack 高可用服务必须建立在 Pacemaker 集群之上。对于 OpenStack 集群而言，控制节点故障意味着 OpenStack 不能提供云计算服务，在本章实现的 OpenStack 高可用集群中，控制节点由三台服务器构成，OpenStack 各个服务以 A/A 或 A/P 高可用模式运行在三个控制节点上，并通过虚拟 IP 地址和负载均衡器对外提供服务，在任一控制节点出现故障时，相应的 OpenStack 服务仍然可以通过负载均衡器继续提供服务。

正常情况下，OpenStack 集群控制层面服务全部运行在三个控制节点上，而 OpenStack 集群计算层面服务全部运行在计算节点上。同时，由于运行 Pacemaker_remote 的 Pacemaker 集群远程节点不具备 Fencing 功能，因此全部 Fencing 设备均运行在控制节点上。现假设其中某个控制节点故障（这里假设 **controller3-vm** 故障），**controller3-vm** 控制节点故障后，Pacemaker 集群中运行的 Resource 和 Stonith 状态如下：

```
[root@controller1-vm ~(keystone_admin)]$ pcs status
Cluster name: openstack-ha
Last updated: Thu Dec 29 11:17:53 2016      Last change: Thu Dec 29 10:02:39
2016 by hacluster via crmd on controller1-vm
Stack: corosync
Current DC: controller1-vm (version 1.1.13-10.el7_2.4-44eb2dd) - partition with quorum
5 nodes and 231 resources configured

Online: [ controller1-vm controller2-vm ]
OFFLINE: [ controller3-vm ]
RemoteOnline: [ computer1 computer2 ]

Full list of resources:

fence1 (stonith:fence_xvm):      Started controller1-vm
fence2 (stonith:fence_xvm):      Started controller1-vm
```

① 也可以将 Openstack 各个服务分散到不同的 Pacemaker 高可用集群中独自实现其高可用。

```

fence3 (stonith:fence_xvm): Started controller2-vm
Clone Set: lb-haproxy-clone [lb-haproxy]
    Started: [ controller1-vm controller2-vm ]
    Stopped: [ computer1 computer2 controller3-vm ]
vip-db (ocf::heartbeat:IPaddr2): Started controller1-vm
vip-rabbitmq (ocf::heartbeat:IPaddr2): Started controller1-vm
vip-keystone (ocf::heartbeat:IPaddr2): Started controller2-vm
vip-glance (ocf::heartbeat:IPaddr2): Started controller2-vm
vip-cinder (ocf::heartbeat:IPaddr2): Started controller1-vm
vip-swift (ocf::heartbeat:IPaddr2): Started controller2-vm
vip-neutron (ocf::heartbeat:IPaddr2): Started controller1-vm
vip-nova (ocf::heartbeat:IPaddr2): Started controller1-vm
vip-horizon (ocf::heartbeat:IPaddr2): Started controller2-vm
vip-heat (ocf::heartbeat:IPaddr2): Started controller2-vm
vip-ceilometer (ocf::heartbeat:IPaddr2): Started controller1-vm
vip-qpid (ocf::heartbeat:IPaddr2): Started controller2-vm
Master/Slave Set: galera-master [galera]
    Masters: [ controller1-vm controller2-vm ]
    Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: memcached-clone [memcached]
    Started: [ controller1-vm controller2-vm ]
    Stopped: [ computer1 computer2 controller3-vm ]
Master/Slave Set: rabbitmq-cluster-master [rabbitmq-cluster]
    Masters: [ controller1-vm ]
    Slaves: [ controller2-vm ]
    Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: mongodb-clone [mongodb]
    Started: [ controller1-vm controller2-vm ]
    Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: keystone-clone [keystone]
    Started: [ controller1-vm controller2-vm ]
    Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: glance-fs-clone [glance-fs]
    Started: [ controller1-vm controller2-vm ]
    Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: glance-registry-clone [glance-registry]
    Started: [ controller1-vm controller2-vm ]
    Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: glance-api-clone [glance-api]
    Started: [ controller1-vm controller2-vm ]
    Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: cinder-api-clone [cinder-api]
    Started: [ controller1-vm controller2-vm ]
    Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: cinder-scheduler-clone [cinder-scheduler]
    Started: [ controller1-vm controller2-vm ]
    Stopped: [ computer1 computer2 controller3-vm ]
cinder-volume (systemd:openstack-cinder-volume): Started controller1-vm
Clone Set: neutron-server-api-clone [neutron-server-api]

```

```

Started: [ controller1-vm controller2-vm ]
Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: neutron-ovs-cleanup-clone [neutron-ovs-cleanup]
Started: [ controller1-vm controller2-vm ]
Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: neutron-netns-cleanup-clone [neutron-netns-cleanup]
Started: [ controller1-vm controller2-vm ]
Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: neutron-openvswitch-agent-clone [neutron-openvswitch-agent]
Started: [ controller1-vm controller2-vm ]
Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: neutron-dhcp-agent-clone [neutron-dhcp-agent]
Started: [ controller1-vm controller2-vm ]
Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: neutron-l3-agent-clone [neutron-l3-agent]
Started: [ controller1-vm controller2-vm ]
Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: neutron-metadata-agent-clone [neutron-metadata-agent]
Started: [ controller1-vm controller2-vm ]
Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: nova-api-clone [nova-api]
Started: [ controller1-vm controller2-vm ]
Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: nova-scheduler-clone [nova-scheduler]
Started: [ controller1-vm controller2-vm ]
Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: nova-conductor-clone [nova-conductor]
Started: [ controller1-vm controller2-vm ]
Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: nova-consoleauth-clone [nova-consoleauth]
Started: [ controller1-vm controller2-vm ]
Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: nova-novncproxy-clone [nova-novncproxy]
Started: [ controller1-vm controller2-vm ]
Stopped: [ computer1 computer2 controller3-vm ]
Master/Slave Set: redis-server-master [redis-server]
Masters: [ controller2-vm ]
Slaves: [ controller1-vm ]
Stopped: [ computer1 computer2 controller3-vm ]
vip-redis (ocf::heartbeat:IPaddr2): Started controller2-vm
Clone Set: ceilometer-central-clone [ceilometer-central]
Started: [ controller1-vm controller2-vm ]
Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: ceilometer-collector-clone [ceilometer-collector]
Started: [ controller1-vm controller2-vm ]
Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: ceilometer-api-clone [ceilometer-api]
Started: [ controller1-vm controller2-vm ]
Stopped: [ computer1 computer2 controller3-vm ]

```

```

Clone Set: ceilometer-delay-clone [ceilometer-delay]
    Started: [ controller1-vm controller2-vm ]
    Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: ceilometer-alarm-evaluator-clone [ceilometer-alarm-evaluator]
    Started: [ controller1-vm controller2-vm ]
    Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: ceilometer-alarm-notifier-clone [ceilometer-alarm-notifier]
    Started: [ controller1-vm controller2-vm ]
    Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: ceilometer-notification-clone [ceilometer-notification]
    Started: [ controller1-vm controller2-vm ]
    Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: heat-api-clone [heat-api]
    Started: [ controller1-vm controller2-vm ]
    Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: heat-api-cfn-clone [heat-api-cfn]
    Started: [ controller1-vm controller2-vm ]
    Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: heat-api-cloudwatch-clone [heat-api-cloudwatch]
    Started: [ controller1-vm controller2-vm ]
    Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: heat-engine-clone [heat-engine]
    Started: [ controller1-vm controller2-vm ]
    Stopped: [ computer1 computer2 controller3-vm ]
Clone Set: horizon-clone [horizon]
    Started: [ controller1-vm controller2-vm ]
    Stopped: [ computer1 computer2 controller3-vm ]
nova-evacuate (ocf::openstack:NovaEvacuate): Started controller1-vm
Clone Set: neutron-openvswitch-agent-compute-clone [neutron-openvswitch-agent-compute]
    Started: [ computer1 computer2 ]
    Stopped: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: libvirt-d-compute-clone [libvirt-d-compute]
    Started: [ computer1 computer2 ]
    Stopped: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: ceilometer-compute-clone [ceilometer-compute]
    Started: [ computer1 computer2 ]
    Stopped: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: nova-compute-fs-clone [nova-compute-fs]
    Started: [ computer1 computer2 ]
    Stopped: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: nova-compute-checkevacuate-clone [nova-compute-checkevacuate]
    Started: [ computer1 computer2 ]
    Stopped: [ controller1-vm controller2-vm controller3-vm ]
Clone Set: nova-compute-clone [nova-compute]
    Started: [ computer1 computer2 ]
    Stopped: [ controller1-vm controller2-vm controller3-vm ]
fence-nova (stonith:fence_compute): Started controller1-vm
computer1 (ocf::pacemaker:remote): Started controller1-vm
computer2 (ocf::pacemaker:remote): Started controller2-vm

```


Controller3-vm 控制节点故障后, Pacemaker 自动将资源分配到 Controller1-vm 和 Controller2-vm 控制节点上继续运行。对于运行在 A/A 高可用模式的服务(如 nova-api), Controller3-vm 的故障不会对服务访问造成影响,但是对于运行在 A/P 高可用模式的服务(如 cinder-volume),如果服务本身在 Controller3-vm 上为 Active,而在其他节点上为 Passive,则 A/P 模式的服务会有短暂的访问中断,而如果服务本身在故障控制节点上为 Passive 状态,则也不会对服务访问造成影响。Controller3-vm 控制节点故障后,在 OpenStack 集群中进行正常的云计算资源请求访问,以验证 OpenStack 仍然可以提供正常的云计算服务,验证过程如下。

检查 Nova 服务运行情况:

```
[root@controller1-vm ~(keystone_admin)]$ nova service-list
```

Id	Binary	Host	Zone	Status	State
2	nova-consoleauth	controller3-vm	internal	enabled	down
5	nova-consoleauth	controller1-vm	internal	enabled	up
8	nova-consoleauth	controller2-vm	internal	enabled	up
11	nova-scheduler	controller1-vm	internal	enabled	up
14	nova-scheduler	controller2-vm	internal	enabled	up
17	nova-scheduler	controller3-vm	internal	enabled	down
20	nova-conductor	controller1-vm	internal	enabled	up
29	nova-conductor	controller2-vm	internal	enabled	up
32	nova-conductor	controller3-vm	internal	enabled	down
41	nova-compute	computer1	nova	enabled	up
44	nova-compute	computer2	nova	enabled	up

检查 Neutron agent 服务运行情况:

```
[root@controller1-vm ~(keystone_admin)]$ neutron agent-list
```

id	agent_type	host	alive	binary
...cbb	DHCP agent	controller3-vm	xxx	neutron-dhcp-agent
...3f4	Metadata agent	controller1-vm	:-)	neutron-metadata-agent
...51e	Open vSwitch agent	computer1	:-)	neutron-openvswitch-agent
...ed2	L3 agent	controller3-vm	xxx	neutron-l3-agent
...aa3	Metadata agent	controller2-vm	:-)	neutron-metadata-agent
...29a	DHCP agent	controller1-vm	:-)	neutron-dhcp-agent
...3a7	Open vSwitch agent	controller1-vm	:-)	neutron-openvswitch-agent
...9dd	Open vSwitch agent	controller3-vm	xxx	neutron-openvswitch-agent
...144	L3 agent	controller2-vm	:-)	neutron-l3-agent
...349	L3 agent	controller1-vm	:-)	neutron-l3-agent
...ab3	Metadata agent	controller3-vm	xxx	neutron-metadata-agent
...5a5	DHCP agent	controller2-vm	:-)	neutron-dhcp-agent
...47e	Open vSwitch agent	controller2-vm	:-)	neutron-openvswitch-agent

```
|...539 | Open vSwitch agent | computer2 | :- ) | neutron-openvswitch-agent |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

查看当前集群中的实例:

```
[root@controller1-vm ~(keystone_admin)]$ nova list
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| ID      | Name      | Status | Task State | Power State | Networks |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|...5f3   | instance1 | ACTIVE | -           | Running     | admin-net=192.128.1.6 |
|...d0a   | instance2 | ACTIVE | -           | Running     | admin-net=192.128.1.7 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

删除实例 instance2:

```
[root@controller1-vm ~(keystone_admin)]$ nova delete instance2
Request to delete server instance2 has been accepted.
```

确认实例 instance2 已被删除:

```
[root@controller1-vm ~(keystone_admin)]$ nova list
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| ID      | Name      | Status | Task State | Power State | Networks |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|...5f3   | instance1 | ACTIVE | -           | Running     | admin-net=192.128.1.6 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

重新创建实例 instance2:

```
[root@controller1-vm ~(keystone_admin)]$ nova boot --image cirros\
--flavor 1 --nic net-id=48274a3a-3222-4d7d-b557-7bbfbalfc70d \
--key-name admin-key --security-group default instance2
[root@controller1-vm ~(keystone_admin)]$ nova list
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| ID      | Name      | Status | Task State | Power State | Networks |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|...5f3   | instance1 | ACTIVE | -           | Running     | admin-net=192.128.1.6 |
|...8c5   | instance2 | ACTIVE | -           | Running     | admin-net=192.128.1.8 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

为实例 instance2 创建 Floating IP:

```
[root@controller1-vm ~(keystone_admin)]$ neutron floatingip-create\
ext-net
Created a new floatingip:
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| Field      | Value |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| fixed_ip_address | 192.168.115.201 |
| floating_ip_address | 192.168.115.201 |
| floating_network_id | 6db6e9b5-98eb-45a0-a00e-271485739fcc |
| id          | b376ac37-6bb8-44f0-b144-aff1dee556de |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

```

| port_id |
| router_id |
| status | DOWN |
| tenant_id | 7de7367105c84b9b822bba79f2c0b80a |
+-----+
[root@controller1-vm ~(keystone_admin)]$ nova floating-ip-associate \
instance2 192.168.115.201
[root@controller1-vm ~(keystone_admin)]$ nova list
+-----+
| ID | Name | Status | Power State | Networks |
+-----+
| ...5f3 | instance1 | ACTIVE | Running | admin-net=192.128.1.6 |
| ...8c5 | instance2 | ACTIVE | Running | admin-net=192.128.1.8, 192.168.115.201 |
+-----+

```

为实例 instance2 挂载 Volume:

```

[root@controller1-vm ~(keystone_admin)]$ cinder list
+-----+
| ID | Status | Display Name | Size | Volume Type | Bootable | Attached to |
+-----+
| ...08d | available | NFS-Volume1 | 2 | - | false | |
+-----+
[root@controller1-vm ~(keystone_admin)]$ nova volume-attach \
instance2 971a0aab-5315-4966-86cd-0dcf2b49c08d auto
+-----+
| Property | Value |
+-----+
| device | /dev/vdb |
| id | 971a0aab-5315-4966-86cd-0dcf2b49c08d |
| serverId | 9aff0d85-5763-49c9-949e-b09a004678c5 |
| volumeId | 971a0aab-5315-4966-86cd-0dcf2b49c08d |
+-----+
[root@controller1-vm ~(keystone_admin)]$ cinder list
+-----+
| ID | Status | Display Name | Size | Volume Type | Bootable | Attached to |
+-----+
| ...08d | in-use | NFS-Volume1 | 2 | - | false | ...8c5 |
+-----+

```

上述操作和验证结果表明，Controller3-vm 控制节点故障后，OpenStack 高可用集群的计算服务（Nova）、镜像服务（Glance）、块存储服务（Cinder）和网络服务（Neutron）等均未受到影响，租户仍然可以继续发出云计算资源相关的服务请求，如创建实例、实例属性设置和网络创建及使用等操作。此外，由于通过虚拟 IP 地址和 HAProxy 负载均衡器对外提供服务，OpenStack 的控制面板服务 Horizon 也不会受到影响，控制节点 Controller3-vm 故障后，经过以上客户端命令行请求，OpenStack 的 Dashboard 服务如图 12-17 所示。

项目

Compute

概況

实例

云硬盘

镜像

访问 & 安全

网络

对象存储

编排

管理员

Identity

实例

云主机名称

镜像名称

IP 地址

配置

值对

状态

instance2	cirros	192.128.1.8 浮动IP: 192.168.115.201	m1.tiny	admin-key	运行中
instance1	cirros	192.128.1.6	m1.tiny	admin-key	关机

正在显示 2 项

图 12-17 OpenStack 高可用集群中的 Dashboard 服务

在本章介绍的 OpenStack 高可用集群部署方案中，全部 OpenStack 开源云计算相关的服务组件均实现了不同层次的高可用保护。结合第 11 章中的基础服务软件高可用部署，截至本章结束，与 OpenStack 开源云计算相关的各个服务均实现了高可用^①，本节的验证结果也充分证实了本章部署实现的 OpenStack 集群具有可靠的高可用性。

12.4 本章小结

基于 Pacemaker 集群的 OpenStack 高可用部署方案，是实际应用中最为常见的高可用部署方案。在 Pacemaker 集群中，OpenStack 的各个服务以集群资源形式接受集群管理控制器 Pacemaker 的统一控制，各服务之间的彼此依赖和启动关系通过 Pacemaker 集群约束来设置。此外，在基于 Pacemaker 集群的 OpenStack 高可用部署中，运行 Pacemaker 和 Csync 的控制节点被称为集群节点，仅运行 Pacemaker_remote 的计算节点被称为远程节点，而集群节点属于控制层面，远程节点则属于计算层面，OpenStack 控制服务运行在控制层面节点上，计算服务运行在计算层面节点上，同时计算层面服务接受控制层面的调度与控制。为了实现 OpenStack 控制服务的高可用，每一项 OpenStack 控制服务均以 A/A 或 A/P 高可用模式运行在控制层面节点上，并通过虚拟浮动 IP 和 HAProxy 负载均衡器对外提供服务，从而任意控制节点故障均不会造成某一 OpenStack 服务或全部 OpenStack 服务的不可用。此外，为了实现 OpenStack 计算服务的高可用，并解决 Pacemaker 集群 16 节点的限制，OpenStack 计算节点仅运行 Pacemaker_remote 集群软件，从而将计算节点加入位于控制层面的 Pacemaker 集群中，并使计算节点成为 Pacemaker 集群的受控远程节点。在实例

① 本章并未覆盖全部“大帐篷”下的所有项目，但是其他项目的高可用部署可以参考本章内容实现。

高可用中，Pacemaker 集群通过 Fence_compute 代理实时监控 Nova-compute 服务运行情况，一旦出现故障便通知 NovaEvacuate 代理进行实例疏散迁移。

本章在第 11 章的基础上，实现了 OpenStack 核心服务组件的高可用部署，并将 OpenStack 服务分为控制层面服务和计算层面服务分别实现高可用部署。此外，本章还对基于 Pacemaker 集群资源管理控制器的 OpenStack 高可用集群进行了部署完成后的功能验证和服务高可用性验证，验证结果充分展示了 Pacemaker 集群中 OpenStack 集群服务的高可用性。本章也是对前文所有章节知识点的综合应用，通过对本章的学习，读者不仅可以独立完成基于生产环境的 OpenStack 高可用集群部署，同时还对本书全部知识点进行了梳理回顾。



运维篇

- 第13章 OpenStack 高可用集群运维最佳实践
 - 第14章 Ceph 存储集群运维最佳实践
-

OpenStack 高可用集群运维最佳实践

OpenStack 本身由诸多复杂的功能组件构成，加之高可用集群资源管理器 Pacemaker 的引入，虽然集群资源管理器使得 OpenStack 及其相关的底层服务具备了高可用性，但也使得高可用集群故障问题的定位分析和排查解决变得更为复杂和困难。在基于 Pacemaker 的 OpenStack 高可用集群中，全部 OpenStack 及其相关底层服务均交由 Pacemaker 以资源形式进行管理调度，并且 Pacemaker 通过各种类型的资源代理脚本对资源进行具体的操作控制，因此，资源代理的运行机制及其调式维护是 OpenStack 高可用集群运维中必须掌握的技能。同时 Pacemaker 由诸多进程组成，集群管理员如何通过各个进程日志判断并排查集群故障，这是 Pacemaker 集群运维中经常面临的问题。此外，由于 OpenStack 上游社区的实例高可用应由集群基础架构自身负责的开发理念，使得实例高可用一直是 OpenStack 集群高可用中的重点和难点，因此如何借用 Pacemaker 实现实例高可用、如何理解实例高可用监控代理脚本以及如何对实例高可用故障问题进行跟踪分析，这些都是 OpenStack 高可用集群运维必须解决的问题。在 OpenStack 的各个核心项目中，Neutron 因其灵活多变的复杂功能，使得其在整个 OpenStack 项目中的复杂程度、理解实现和运行维护都是极为头痛的事情。本章中，上述基于 Pacemaker 的 OpenStack 高可用集群经常面临的各种运维问题都会被描述介绍，同时还对运行维护中经常碰到的问题、难题和各种杂症进行了原理分析与解决方案的介绍，并且对 OpenStack 日常运行中经常面对的各种操作进行了归纳总结，本章内容可以归纳为对 Pacemaker 和 OpenStack 高可用集群运行维护的参考实践。

13.1 Pacemaker OCF 资源代理故障诊断分析

在基于 Pacemaker 的 OpenStack 高可用集群部署实施及运行维护中，最主要的维护工作便是确保集群资源的正常运行及其高可用性，因为全部 OpenStack 核心项目服务及其所依赖的基础软件服务均以资源对象的形式在集群中以各种高可用模式运行，以本书介绍的 OpenStack 高可用部署方案为例，Pacemaker 集群实验环境由三个控制节点和两个计算节点以及两百多个资源组成，随着 OpenStack 部署项目的增多，Pacemaker 集群资源数目将不断增加，因此熟悉和掌握 Pacemaker 集群中不同的资源类型及其管理方式，对 OpenStack 高可用集群的运行维护将起到事半功倍、化简为繁的效果。

13.1.1 Pacemaker 集群 OCF 资源代理使用介绍

在 Pacemaker 集群中，“资源”实际上并没有任何官方的正式定义，而是认为由集群负责管理的任何对象都称之为资源^①，资源由资源代理（Resource Agent，RA）负责管理，RA 是管理集群资源的可执行文件。在 Linux 系统中，资源代理有 OCF、LSB、Systemd 和 Stonith 等类型之分，其中 LSB 类型 RA 是传统 Linux 系统中最为常见的资源代理，通常位于 /etc/init.d 目录中的脚本便是 LSB 类型的 RA，随着各个 Linux 新版本的发布，Systemd 类型 RA 已逐渐替代了 LSB，如在 RHEL7 和 Centos7 版本中，系统服务的启停控制已不再由 /etc/init.d 脚本实现，而由 Systemd 以单元文件形式控制。除了 LSB/Systemd 类型 RA，Pacemaker 集群中最为常用的资源代理便是 OCF 类型 RA，OCF 是对传统 LSB 类型 RA 的扩展。相对 LSB，符合开放集群框架标准的 OCF 类型 RA 具有更多的行为参数以及对 RA 进行功能和参数描述的元数据信息，在 Linux 系统中，任何符合 OCF 标准^②的程序文件都可以称为 OCF 类型 RA，因此任何用户或系统管理员均可按照 OCF 规范使用各种编程语言定义自己的 OCF 资源代理，由于 OCF 与主流集群资源管理器（如 Pacemaker 和 RGmanager）兼容，因此用户定义的 OCF 脚本可由 Pacemaker 资源管理器直接使用。

作为开源云计算的主流，OpenStack 不仅对源代码开源，还有完善的 API 接口供二次开发使用，这为终端用户和运维管理员自定义 OCF 和 Stonith 等类型 RA 以实现集群管理和自动化运维提供了极大便利。在基于 Pacemaker 的 OpenStack 高可用集群中，由于社区并未提供实例 HA 功能，因此计算节点实例 HA 方案需要用户结合 OpenStack API 接口和集群管理基础软件共同实现，其中对计算节点的故障监控、节点隔离和实例恢复便是由 Stonith 和 OCF 类型资源代理来实现的。在实际的计算节点高可用部署中，使用的便是 Stonith 类型资源代理 fence_compute 和 OCF 类型资源代理 NovaEvacuate，而在 OpenStack 控制节点服务高可用部署中，还会使用除了 NovaEvacuate 外的多个 OCF 类型 RA。在 RHEL 或 CentOS 发行版的 Linux 系统中，很多常用的 OCF 类型 RA 被集成在 Resource-agents 安装包中，安

① <http://www.linux-ha.org/doc/dev-guides/ra-dev-guide.html>

② <http://www.opencf.org/cgi-bin/viewcvs.cgi/specs/ra/resource-agent-api.txt?rev=HEAD>

装后存放在 /usr/lib/ocf/resource.d/heartbeat 目录中，如果用户需要自定义 OCF 脚本，则脚本的最终存放路径应该位于 /usr/lib/ocf/resource.d/\$provider_name 中，其中 \$provider_name 为用户自定义的 OCF 类型 RA 的 Provider 名称，如 OpenStack。在 Pacemaker 集群中，通过 pcs 的 resource 命令即可查看与 OCF 相关的 RA，如查看当前系统中提供了 OCF 类型 RA 的全部 Provider，查看命令如下：

```
[root@controller1-vm ~]# pcs resource providers
heartbeat
neutron
openstack
pacemaker
rabbitmq
```

上述系统输出的 Provider 中，除了 heartbeat 为操作系统的 Resource-agents 安装包提供外，其余全部为用户后续自定义或独立安装的 Provider，如提供 Neutron 网络服务高可用的 Provider 为 neutron、提供 OpenStack 实例 HA 的 Provider 为 OpenStack 以及提供 RabbitMQ 高可用的 Provider 为 rabbitmq。要查看当前系统中 Pacemaker 可以使用的全部 OCF 类型 RA，可通过如下命令实现：

```
[root@controller1-vm ~]# pcs resource agents ocf
CTDB
Delay
Dummy
Filesystem
IPaddr
IPaddr2
IPsrcaddr
LVM
...
```

上述命令显示了全部 OCF 类型 Provider 子目录下的 RA 文件，这些 RA 文件分别由 heartbeat、neutron、openstack、pacemaker 和 rabbitmq 等 Provider 提供，要查看特定 Provider 提供的 RA，可通过如下命令实现：

```
//Neutron提供的RAS
[root@controller1-vm ~]# pcs resource agents ocf:neutron
NetnsCleanup
NeutronScale
OVSCleanup
//Openstack提供的RAS
[root@controller1-vm ~]# pcs resource agents ocf:openstack
NovaEvacuate
nova-compute-wait
rabbitmq提供的RAS
[root@controller1-vm ~]# pcs resource agents ocf:rabbitmq
rabbitmq-server
rabbitmq-server-ha
set_rabbitmq_policy.sh
```

13.1.2 Pacemaker 集群 OCF 资源代理定义语法

由于 Pacemaker 使用很多 OCF 类型 RA 管理各种高可用资源，熟悉掌握符合 OCF 标准的资源代理脚本语法、变量、调式以及执行过程和运行状态，对于用户自定义实现特定功能的 OCF 脚本和排除他人提供的 OCF 脚本故障会有很大帮助，因此本节将重点介绍 OCF 标准语法、变量传递及脚本调式等内容，从而为读者在 Pacemaker 集群中使用或自定义 OCF 脚本时提供帮助。本节将从 OCF 资源代理脚本的 API 定义、返回代码、RA 结构、RA 操作、脚本变量和标准函数几个方面来介绍 OCF 脚本。

1. API 定义

OCF 资源代理通过环境变量获取与其所管理的集群资源相关的配置变量信息，环境变量总是以资源变量参数加上“OCF_RESKEY_”前缀命名，例如 nova-evacuate 资源在创建时指定的资源参数如下：

```
pcs resource create nova-evacuate ocf:openstack:NovaEvacuate \
auth_url=http://$vip_keystone:35357 username=admin password=admin \
tenant_name=admin --force
```

其中，auth_url、username、password 和 tenant_name 均为 nova-evacuate 资源参数，而在 NovaEvacuate 资源代理脚本中，获取以上参数值时应该在资源变量名前补上前缀，因此对应的变量为 OCF_RESKEY_auth_url、OCF_RESKEY_username、OCF_RESKEY_password 和 OCF_RESKEY_tenant_name。对于某一个资源参数，如果其并非为必须的参数，即 RA 元数据定义中“required=0”的参数，RA 脚本必须为其提供一个合理的默认值，而 RA 通过 OCF_RESKEY_<parametername>_default 来获取该参数的默认值。此外，任何 RA 都必须提供一个命令行参数，该参数用以告知 RA 应该执行什么操作，而 RA 必须支持的操作参数包括以下四个：

- ❑ start，启动资源；
- ❑ stop，停止资源；
- ❑ monitor，查询获取资源运行状态；
- ❑ meta-data，输出资源代理的元数据信息。

除了上述四个必须的资源代理操作参数，RA 还可以选择性地支持其他几个操作参数，这些操作参数主要用以进行高级资源操作。RA 可选择性支持的操作参数如下：

- ❑ promote，在 Master/Slave 多状态资源中将资源提升为 Master；
- ❑ demote，在 Master/Slave 多状态资源中将资源降级为 Slave；
- ❑ migrate_to&migrate_to，实现资源的 live-migration；
- ❑ validate-all，验证资源配置信息是否正确；
- ❑ usage/help，当用户从命令行调用 RA 时显示资源使用信息；
- ❑ status，已被 monitor 替代，二者功能相同。

每个 RA 资源代理的操作行为 (Action) 有对应的超时 (Timeout) 时间, 集群管理器负责监控 RA 的某个 Action 已经执行了多长时间, 而 RA 本身无须监控 Action 的执行时间, 如果在 Timeout 规定的时间内 Action 的结果未能达到预设状态, 则集群管理器将终止当前 Action。通常, 为了便于最终用户使用, RA 会在 metadata 定义部分给出针对不同 Action 的 Timeout 建议值, 用户在创建资源时可以参考此值设置不同 Action 的 Timeout 值。在 RA 脚本中, 为了便于用户理解使用, 每个 RA 必须在 XML 元数据集中说明它的用途以及支持使用的参数, 集群管理器的在线帮助命令和关于此 RA 的 man 帮助命令将会从此 XML 元数据中提取帮助信息, RA 的 XML 元数据定义格式如下:

```
</resource-agent>
.....
<parameter>
<parameter name="datadir" unique="0" required="1">
<longdesc lang="en">
    Data directory, an example string parameter
</longdesc>
<shortdesc lang="en">Data directory</shortdesc>
<content type="string"/>
</parameter>
</parameters>
<actions>
<action name="start"          timeout="20" />
<action name="stop"           timeout="20" />
<action name="monitor"        timeout="20"
interval="10" depth="0" />
<action name="reload"         timeout="20" />
<action name="migrate_to"     timeout="20" />
<action name="migrate_from"   timeout="20" />
<action name="meta-data"      timeout="5" />
<action name="validate-all"   timeout="20" />
</actions>
</resource-agent>
```

在 XML 元数据中定义参数时, required 代表该参数是否是必须的, 如 “required=1” 则表示必须的, 而 “required=0” 则表示可选的。unique 代表该参数在整个集群中是否唯一, “unique=1” 表示唯一, “unique=0” 则表示可重复。元数据的 actions 定义段给出了此 RA 支持的 Action, 每个 Action 都应给出 Timeout 时间, 这是向用户提供的此 Action 最小超时建议时间, 因为不同的资源针对此操作时可能需要不同的时间, 如 IP 地址或文件系统资源针对 start 操作就很快, 而数据库资源则需要几分钟。此外, 一些重复性操作 (如 monitor) 还应该提供操作执行的最小间隔时间。

2. 返回代码

在 RA 中, 任何调用都应该返回一个预定义的代码, 以告知调用者操作执行的结果。在 OCF 类型的 RA 中, 每个返回代码都有一个变量名与之对应, 在 RA 脚本内部, 每个调

用函数在执行完毕后，都以 `return` 返回执行结果，而执行结果则由返回代码来表示。除了使用 `return` 返回执行结果，在 RA 脚本的执行过程中如果遇到严重错误，还可以使用 `exit` 直接退出 RA 程序执行，同时给出程序意外终止的原因，终止原因也由返回代码来表示。如下代码片段是 NovaEvacuate 代理程序中使用 `return` 和 `exit` 进行返回和退出的例子：

```
.....
if [ $? != 0 ]; then
    ocf_exit_reason "Invalid state directory: $state_dir"
    return $OCF_ERR_ARGS
fi
if [ -z "${OCF_RESKEY_auth_url}" ]; then
    ocf_exit_reason "auth_url not configured"
    exit $OCF_ERR_CONFIGURED
fi
.....
```

在 OCF 规范中，针对不同执行结果和错误类型，均有对应的代码和变量名与之对应，在 RA 脚本内部，使用变量名作为返回或退出结果，而在 Pacemaker 日志文件中，针对不同的操作结果显示的是对应的数值代码。OCF 中的返回代码与变量名称的对应关系及代码解释如表 13-1 所示。

表 13-1 OCF 资源代理返回代码对照表

变量名称	代 码	返回结果说明
OCF_SUCCESS	0	操作执行成功
OCF_ERR_GENERIC	1	操作返回常规错误，不能使用其他代码精确定位时才使用
OCF_ERR_ARGS	2	调用 RA 的参数不正确，通常表明没有按照 RA 支持的参数使用
OCF_ERR_UNIMPLEMENTED	3	RA 未实现正在执行的 Action
OCF_ERR_PERM	4	没有充分的权限执行 RA
OCF_ERR_INSTALLED	5	由于缺少必须的组件，Action 在当前节点执行失败
OCF_ERR_CONFIGURED	6	资源参数配置错误，如给字符串类型变量赋值整数
OCF_NOT_RUNNING	7	RA 所管理的资源未处于运行状态
OCF_RUNNING_MASTER	8	资源运行在 Master 状态，只能由 Monitor 操作返回
OCF_FAILED_MASTER	9	资源不能处于 Master 状态，只能由 Monitor 操作返回

3. RA 程序结构

对于 OCF 类型的 RA 脚本程序，其内容构成有着标准的模块结构，这里以最为常见的

shell 脚本 RA 为例，一个标准的 OCF 类型 RA 程序段通常由脚本解释器、作者及许可信息、RA 初始化、针对 RA 不同 Action 的函数实现以及位于最后的可执行程序块（类似于程序的 main 函数）构成。作为一个可执行的脚本，任何 RA 程序必须在第一行以“#!”符号给出脚本执行时使用的解释器，如下：

```
//sh解释器
#!/bin/sh
//bash解释器
#!/bin/bash
```

通常，对于由 shell 语言编写的 RA，sh 是推荐使用的常规解释器，但是对于某些特殊的变量定义及引用方式，也可以使用 bash 解释器。在解释器之后，应该跟上与此 RA 相关的作者和许可信息，如下是 NovaEvacuate 资源代理的版权及许可信息：

```
# NovaCompute agent manages compute daemons.
# Copyright (c) 2015
# This program is free software; you can redistribute it and/or modify
# it under the terms of version 2 of the GNU General Public License as
# published by the Free Software Foundation.
.....
```

作者及许可信息之后，是 RA 程序的初始化部分，在正式编写 RA 脚本之前，必须将 ocf-shellfuncs 函数库加载到当前环境中，加载过程便是由初始化语句完成。基于 shell 脚本的 OCF 类型 RA 初始化语句通常为如下形式：

```
: ${OCF_FUNCTIONS_DIR:=${OCF_ROOT}/lib/heartbeat}
. ${OCF_FUNCTIONS_DIR}/ocf-shellfuncs
```

初始化完成之后，接着是对 RA 支持的各个 Action 的函数实现，通常 RA 提供的每个 Action 都对应着一个函数实现，如 Start、Stop 和 Monitor 等 Action 都需要专门的函数对其进行实现。在 RA 的最后，是 RA 被调用时真正执行的代码部分，通常称为可执行块（Execution Block），可执行块代码遵循相对标准的结构，不同 RA 的可执行块几乎完全类似。如下是 NovaEvacuate 资源代理的可执行块：

```
case $__OCF_ACTION in
start)      evacuate_validate; evacuate_start;;
stop)       evacuate_stop;;
monitor)    evacuate_validate; evacuate_monitor;;
meta-data)  meta_data
            exit $OCF_SUCCESS
            ;;
usage|help) evacuate_usage
            exit $OCF_SUCCESS
            ;;
validate-all) exit $OCF_SUCCESS;;
*)          evacuate_usage
            exit $OCF_ERR_UNIMPLEMENTED
            ;;
```

```

esac
rc=$?
ocf_log debug "${OCF_RESOURCE_INSTANCE} $__OCF_ACTION : $rc"
exit $rc

```

可执行块的主要作用就是接收 RA 调用者传递进来的 Action 参数，并根据不同的 Action 参数选择执行不同的函数，最终返回 RA 调用的执行结果。

4. RA Action

在 RA 脚本程序中，每个 Action 都作为一个独立的函数或方法而被实现，方便起见，针对每个 Action 的实现函数通常以 `<agent>_<action>` 命名，例如在 NovaEvacuate 资源代理中，针对 Start 操作的 Action 函数命名为 `evacuate_start()`。在 Action 函数实现中，当 RA 碰到不可恢复的错误时，可以在函数体的内部对本次 RA 调用进行立即退出、抛出异常和停止执行等操作，Actions 函数内部可能碰到的错误或异常包括配置问题、权限问题和依赖缺失等，作为通常使用的规则，不建议将这些异常或错误沿调用堆栈往上传递，当某个节点上的 RA 操作（如 Start 和 Monitor）被异常终止后，对于资源的 Start 和 Monitor 等操作的执行应该由集群资源管理器根据用户的资源创建和配置过程进行恢复（如在其他节点上重新执行失败的 RA 操作），而不应该由 RA 自行诊断。在 OCF 类型 RA 中，较为常见的 Action 有以下几种：

- ❑ **Start Action**：当 RA 被以 Start Action 调用时，如果资源处于停止状态，则 RA 必须将资源启动，资源启动之前，RA 必须验证资源配置、查询资源状态，然后启动未运行的资源，在以 Start Action 调用 RA 之前，最常见的做法就是先调用 `validate_all` 和 `monitor` 函数对资源进行配置验证和状态查询。
- ❑ **Stop Action**：当 RA 被以 Stop Action 调用时，如果资源处于运行状态，则 RA 必须迫使资源停止运行。在以 Stop Action 调用 RA 之前，通常先调用 `validate_all` 和 `monitor` 函数对资源配置进行验证和状态查询，然后再迫使运行中的资源停止。对于 Stop Action，正常的返回代码应该是代表执行成功的 `$OCF_SUCCESS`，而不是代表资源未运行的 `$OCF_NOT_RUNNING`。
- ❑ **Monitor Action**：当以 Monitor Action 调用 RA 时，RA 将查询资源状态。Monitor 查询到的资源状态主要有三种，即资源当前正在运行（返回 `$OCF_SUCCESS`）、资源已正常停止（返回 `$OCF_NOT_RUNNING`）和资源处于故障状态，当资源处于故障状态时，RA 将认为资源运行失败，并根据问题原因返回相关的 `$OCF_ERR_xxx` 代码。
- ❑ **Validate-all Action**：Validate-all 主要用于测试资源配置和运行环境的正确性，通常并不直接调用 Validate-all 函数，而是在调用 Start、Stop 和 Monitor 等 Action 函数时，为了验证资源配置信息而间接性地调用 Validate-all 函数。
- ❑ **Meta-data Action**：Meta-data 函数主要用于定义 RA 的 XML 形式元数据信息，调用

该函数时将在标准输出中显示 RA 相关的元数据信息。

- ❑ **Promote Action** : Promote Action 不是常规 RA 必须支持的操作,但是,需要识别 Master 和 Slave 状态的有状态 (stateful) RA 必须实现对 Promote Action 的支持。有状态 RA 的 Slave 状态其实等同于常规无状态 RA 的 Started 状态,因此常规无状态 RA 只需实现 Start 和 Stop Action,而有状态 RA 还必须实现能够在 Started/Slave 与 Master 之间进行资源状态转换的 Promote 操作。
- ❑ **Demote Action** : Demote 与 Promote 实现的功能相反,有状态 RA 中的 Demote 操作将资源从 Master 状态转换为 Slave/Started 状态。

5. RA 脚本变量

为了便于使用,OCF 标准中定义了部分脚本变量,用户在定义自己的 OCF 类型 RA 脚本时,可以直接引用这些变量。常使用的 OCF 脚本变量及其变量值如下:

- ❑ **\$OCF_ROOT** : \$OCF_ROOT 变量是 OCF 脚本存放路径的根目录,用户在 RA 脚本中不能更改此变量值,\$OCF_ROOT 变量值通常默认为 /usr/lib/ocf。
- ❑ **\$OCF_FUNCTIONS_DIR** : \$OCF_FUNCTIONS_DIR 变量代表了 OCF 类型 RA 的 shell 函数库 “.ocf-shellfuncs” 的存放目录,该变量值通常由 \$OCF_ROOT 变量来定义,因此 RA 脚本不能更改此变量值,但是在测试 RA 时可以通过命令行覆盖此变量值。
- ❑ **\$OCF_RESOURCE_INSTANCE** : \$OCF_RESOURCE_INSTANCE 变量代表资源实例的名称,对于原始资源 (即未定义为克隆和有状态的资源),该变量的值就是资源名称,而对于克隆和有状态资源,则是在原始资源名称后跟上冒号和克隆实例的编号,如 Keystone:0。
- ❑ **\$__OCF_ACTION** : \$__OCF_ACTION 变量表示当前调用 RA 的 Action 参数,其值为资源管理器调用 RA 时,指定的第一个命令行参数。
- ❑ **\$__SCRIPT_NAME** : \$__SCRIPT_NAME 变量代表 RA 脚本名称,如位于 /usr/lib/ocf/resource.d/heartbeat 目录中的文件名。
- ❑ **\$SHA_RSCTMP** : \$SHA_RSCTMP 变量代表 RA 脚本执行过程中使用到的临时目录,节点重启后该临时目录中的内容将被清空,因此该临时目录在节点重启后不会保存任何状态数据。

6. RA 标准函数

在 OCF 类型 RA 脚本中,用户可以直接使用很多已经封装好的标准函数,通过这些函数可以实现很多常见和重复的功能,如日志记录和参数判断等。较常使用的 OCF 标准函数有以下几个:

(1) ocf_log 函数

ocf_log 是封装好的库函数,OCF 脚本使用其进行日志记录,ocf_log 函数的使用语法

如下：

```
ocf_log <severity> "Log message"
```

其中，severity 表示记录日志的严重级别，如 debug、info、warn、err 和 crit，在这几个级别的日志记录中，debug 和 info 是输出信息最多的记录方式，通常在 RA 的测试阶段使用，而 err 和 crit 是级别很高的日志记录方式，通常仅在发生异常或错误时才记录日志信息。

(2) have_binary 与 check_binary

通常 RA 脚本需要测试某个需要使用到的可执行文件是否存在，此时 have_binary 与 check_binary 便可方便地实现此功能，have_binary 的使用方法如下：

```
if ! have_binary fence_compute; then
ocf_log warn "Missing fence_compute binary, Please install \
fence-agents-compute!"
fi
```

如果 have_binary 发现指定的可执行文件并不存在，则可以向用户给出提示，但是不会造成 RA 执行中断。如果缺失的可执行文件对 RA 的执行至关重要，则使用 check_binary 来判断，在未发现指定的可执行文件时，check_binary 将中断 RA 的运行并返回代表依赖缺失的代码，即 \$OCF_ERR_INSTALLED。check_binary 的使用很简单，如要成功执行 RA 则 fence_compute 必须存在，可以使用如下命令进行异常中断：

```
check_binary fence_compute
```

(3) ocf_run

当 RA 需要执行某个命令并捕捉其输出时，则需要使用 ocf_run 函数。在使用 ocf_run 函数时，需要执行的命令以字符串参数的形式传递给 ocf_run，如下：

```
ocf_run "frobnicate --spam=eggs" || exit $OCF_ERR_GENERIC
```

如果命令执行成功，则 ocf_run 使用 info 级别记录日志信息，而如果命令执行失败，则使用 err 级别记录日志。

(4) ocf_is_decimal

数值变量判断函数，当 RA 需要验证某个参数值是否是数值时，便可使用此函数实现，如果参数值为数值则返回 True，否则返回 False，ocf_is_decimal 函数的使用方式如下：

```
.....
foobar_validate_all() {
if !ocf_is_decimal $OCF_RESKEY_eggs; then
ocf_log err "eggs is not numeric!"
exit $OCF_ERR_CONFIGURED
fi
...
}
```

(5) ocf_is_true

布尔值判断函数，当 RA 需要判断某个参数值是否为 True/1/on 时，使用此函数即可方便实现，如果参数值为 True/1/on 则返回 True，否则为 False，ocf_is_true 函数的使用如下：

```
if ocf_is_true $OCF_RESKEY_superfrobinate; then
    ocf_run "frobinate --super"
fi
```

13.1.3 Pacemaker 集群 OCF 资源代理调试诊断

对于初次使用他人提供的 OCF 资源代理来配置使用 Pacemaker 资源的用户，尤其是对于 Action 函数的实现较为复杂的 RA，对 Pacemaker 是如何调用 OCF 资源代理程序，以及 OCF 程序的执行过程总是很困惑，以至于在不能实现 RA 声明的功能时不知所措，本节将介绍如何对已测试发行的 RA 进行常规调试跟踪，从而对 RA 的执行过程进行观察并诊断定位可能的故障点。在基于 Pacemaker 的 OpenStack 高可用集群中，NovaEvacuate 资源代理是 OCF 类型的 RA，其主要作用在于跟踪 OpenStack 计算节点的故障状态并触发 Nova API 的调用以迁移故障计算节点上的实例，本节将以 NovaEvacuate 资源代理为例讲解 Pacemaker 集群资源管理器如何调用 OCF 类型 RA 以及如何跟踪调试 RA 的执行过程。Pacemaker 集群中，NovaEvacuate 资源代理所管理资源的创建配置过程如下：

```
pcs resource create nova-evacuate ocf:openstack:NovaEvacuate \
auth_url=http://$vip_keystone:35357 username=admin password=admin \
tenant_name=admin --force
```

上述 pcs 命令创建了一个名为 nova-evacuate 的资源（资源名称可以自定义），创建过程中 NovaEvacuate 必须使用的参数（RA 的 Metadata 中对参数进行了定义）被赋予了特定的参数值。创建完成后，通过 pcs 命令可以查看与 nova-evacuate 资源相关的属性，如下：

```
[root@controller2-vm ~]# pcs resource show nova-evacuate
Resource: nova-evacuate (class=ocf provider=openstack type=NovaEvacuate)
Attributes: auth_url=http://192.168.142.203:35357/v2.0/ username=admin
            password=admin
            tenant_name=admin
Operations: start interval=0s timeout=20 (nova-evacuate-start-interval-0s)
            stop interval=0s timeout=20 (nova-evacuate-stop-interval-0s)
            monitor interval=10 timeout=600 (nova-evacuate-monitor-interval-10)
```

pcs 资源查看命令输出信息中，资源 Resource 信息表明 nova-evacuate 资源的类型为 OCF，Provider 为 OpenStack，Type（即 RA）为 NovaEvacuate。Attributes 信息显示了资源运行时的参数，参数值由资源创建时指定，当 Pacemaker 对 nova-evacuate 资源进行管理时这些参数被传递给 NovaEvacuate 资源代理脚本。Operations 信息显示了与此资源相关的操作及其超时和间隔时间信息，此处仅显示了必须的 Start、Stop 和 Monitor 操作信息，Timeout

和 Interval 时间值是 NovaEvacuate 的默认值。在 RHEL7 或 CentOS7 中, Pacemaker 默认的日志信息文件为 /var/log/cluster/corosync.log, 因此与 nova-evacuate 资源相关的 Pacemaker 管理日志将被定向到该文件中。当通过 Pacemaker 的 pcs 命令重启 nova-evacuate 资源时 (pcs resource restart nova-evacuate), 与 nova-evacuate 资源相关的 Pacemaker 日志信息如图 13-1 所示。

在图 13-1 的 Pacemaker 日志中, 可以看到 Pacemaker 各个进程对 nova-evacuate 资源进行操作的日志信息, 但是 Pacemaker 日志仅体现了对 nova-evacuate 资源的操作过程, 并未输出 NovaEvacuate 资源代理程序的 debug 或 info 级别信息, 因此如果对 Pacemaker 各个进程工作机制不是非常熟悉, 在资源故障时仅通过 Pacemaker 日志是难以诊断问题的, 通常需要对负责资源管理的 RA 进行调式跟踪, 这样方可分析出 RA 程序的执行逻辑, 从而进一步定位资源问题故障点。对于 OCF 类型的 RA, 通常可以通过两种方式对其进行调式运行, 一种是编辑 RA 脚本使其在接受 Pacemaker 调用时将日志重定向到独立日志文件中, 另一种是手工执行 RA 脚本以观察其执行情况。

```
[root@controller3-va ~]# tail -f /var/log/cluster/corosync.log | grep -i nova-evacuate
Jan 05 14:18:50 [1447] controller3-va cib: info: cib_perform_op: ++ /cib/configuration/resources/primitive[@id='nova-evacuate']/meta:
Jan 05 14:18:50 [1458] controller3-va crad: info: abort_transition_graph: Transition aborted by nova-evacuate-meta: attributes-target-rc
Jan 05 14:18:50 [1451] controller3-va stonith-ng: info: update_cib_stonith_devices_v2: Updating device list from the cib: create meta_attrit
Jan 05 14:18:50 [1457] controller3-va pengine: info: native_print: nova-evacuate (ocf:openstack:NovaEvacuate): (target-role:stopped)
Jan 05 14:18:50 [1457] controller3-va pengine: info: native_color: Resource nova-evacuate cannot run anywhere
Jan 05 14:18:50 [1457] controller3-va pengine: notice: LogActions: Stop nova-evacuate (controller2-va)
Jan 05 14:18:50 [1458] controller3-va crad: notice: te_rsc_command: Initiating action 613: stop nova-evacuate_stop_0 on controller2-va
Jan 05 14:18:51 [1447] controller3-va cib: info: cib_perform_op: + /cib/status/node_state[@id='2']/lra[@id='2']/lra_resources/lra_res
Jan 05 14:18:51 [1458] controller3-va crad: info: match_graph_event: Action nova-evacuate_stop_0 (613) confirmed on controller2-va (rc=0)
Jan 05 14:18:54 [1447] controller3-va cib: info: cib_perform_op: -- /cib/configuration/resources/primitive[@id='nova-evacuate']/meta:
Jan 05 14:18:54 [1458] controller3-va crad: info: abort_transition_graph: Transition aborted by deletion of nvpair[@id='nova-evacuate-
Jan 05 14:18:54 [1451] controller3-va stonith-ng: info: stonith_device_remove: Device 'nova-evacuate' not found (4 active devices)
Jan 05 14:18:54 [1457] controller3-va pengine: info: native_print: nova-evacuate (ocf:openstack:NovaEvacuate): Stopped
Jan 05 14:18:54 [1457] controller3-va pengine: info: RecurringOp: Start recurring monitor (10s) for nova-evacuate on controller2-va
Jan 05 14:18:54 [1458] controller3-va crad: notice: LogActions: Start nova-evacuate (controller2-va)
Jan 05 14:18:54 [1447] controller3-va cib: notice: te_rsc_command: Initiating action 612: start nova-evacuate_start_0 on controller2-va
Jan 05 14:18:55 [1458] controller3-va crad: info: cib_perform_op: + /cib/status/node_state[@id='2']/lra[@id='2']/lra_resources/lra_res
Jan 05 14:18:55 [1458] controller3-va crad: info: match_graph_event: Action nova-evacuate_start_0 (612) confirmed on controller2-va (rc=0)
Jan 05 14:18:55 [1458] controller3-va crad: notice: te_rsc_command: Initiating action 613: monitor nova-evacuate_monitor_10000 on control
Jan 05 14:18:55 [1447] controller3-va cib: info: cib_perform_op: + /cib/status/node_state[@id='2']/lra[@id='2']/lra_resources/lra_res
Jan 05 14:18:55 [1458] controller3-va crad: info: match_graph_event: Action nova-evacuate_monitor_10000 (613) confirmed on controller2-va
```

图 13-1 nova-evacuate 资源在 Pacemaker 日志中的信息

1. RA 日志重定向

在 Linux 系统中, OCF 类型 RA 脚本位于 /usr/lib/ocf/resource.d/\$provider_name 目录中, 而且管理员可以重新编辑 RA 脚本, 由于大多数 RA 由 shell 解释性语言编写, 因此只需简单的修改保存即可使用。为了对 RA 的执行过程进行跟踪, 可以在 RA 脚本中不同的地方放置重定向输出语句, 使其在执行过程中将跟踪信息重定向到独立日志文件中。简单起见, 可以将反复使用的重定向语句定义在 RA 的内部函数中, 以 NovaEvacuate 资源代理为例, 可以在 /usr/lib/ocf/resource.d/openstack/NovaEvacuate 文件中新增如下函数:

```
function log_info ()
{
    DATE_N=`date "+%Y-%m-%d %H:%M:%S"`
    USER_N=`whoami`
    echo "${DATE_N} ${USER_N} [INFO] ${1}" >>/var/log/evacuate.log
}
```


log_info 函数会将运行 RA 的用户和时间信息随日志一起重定向至 /var/log/evacuate.log 文件中。而在 RA 的其他需要输出日志的地方，只需简单地调用 log_info 函数即可，如下：

```
.....
//NovaEvacuate的Stop函数
evacuate_stop() {
rm -f "$statefile"
log_info "NovaEvacuate agent stop successfull,directory $statefile removed!OCF state
is $OCF_SUCCESS"
return $OCF_SUCCESS
}
//NovaEvacuate的Start函数
evacuate_start() {
log_info "2.Start NovaEvacuate!directory $statefile will be created!"
touch "$statefile"
return $?
}
.....
```

用户可以自定义 log_info 函数的消息体 (Messages Body)，这些消息在 Pacemaker 调用 NovaEvacuate 资源代理时都会被重定向至 /var/log/evacuate.log 文件中。NovaEvacuate 的可执行块如下所示；

```
.....
case $__OCF_ACTION in
start)          evacuate_validate; evacuate_start;;
stop)           evacuate_stop;;
monitor)        evacuate_validate; evacuate_monitor;;
meta-data)      meta_data
exit $OCF_SUCCESS
;;
usage|help)     evacuate_usage
exit $OCF_SUCCESS
;;
validate-all)  exit $OCF_SUCCESS;;
*)              evacuate_usage
exit $OCF_ERR_UNIMPLEMENTED
;;
esac
rc=$?
ocf_log debug "${OCF_RESOURCE_INSTANCE} $__OCF_ACTION : $rc"
exit $rc
```

其中，最关键的三个 Action 分别为 Start、Stop 和 Monitor，而在 Pacemaker 以 Start 和 Monitor 操作调用 NovaEvacuate 时，都会调用 evacuate_validate 函数对资源配置进行验证，因此在 Pacemaker 启动 nova-evacuate 资源时，evacuate_validate 将会是 NovaEvacuate 执行的第一个函数，而 evacuate_start 是第二个被执行的函数，资源状态查询函数 evacuate_monitor

是第三个被执行的函数。而在 nova-evacuate 启动完成后, NovaEvacuate 将在 evacuate_validate 和 evacuate_monitor 之间重复执行, 根据 NovaEvacuate 的 Monitor 操作时间间隔, 重复时间为 10s。图 13-2 是针对 NovaEvacuate 脚本进行日志自定义和重定向后, Pacemaker 启动 nova-evacuate 资源, 且资源在控制节点 controller2-vm 正常运行时, 对其进行重启操作 (pcs resource restart nova-evacuate) 的日志输出。

从图 13-2 中可以看到, 当 Pacemaker 对 nova-evacuate 进行 restart 操作后, NovaEvacuate 首先执行 evacuate_stop 函数, 之后再执行 evacuate_validate 函数和 evacuate_start 函数。当 nova-evacuate 启动完成后, NovaEvacuate 在 evacuate_validate 与 evacuate_monitor 函数之间重复执行以监控 nova-evacuate 的运行状态。根据用户自己的调试需求, 可以在 NovaEvacuate 中自定义更为详细的调试输出信息, 而根据在 RA 中不同函数内部置入的调试信息, 用户不仅可以在资源正常运行时监控 RA 的行为逻辑, 还可以在资源故障时通过 RA 的重定向日志, 分析可能引起资源异常的故障点, 从而进一步分析故障原因。

```
[root@controller2-vm ~]# tail -f /var/log/evacuate.log
2017-01-05 22:15:56 root [INFO] 1.begining evacuate_validate!
2017-01-05 22:15:56 root [INFO] the fence_options is: -k http://192.168.142.203:35357/v2.0/ -l admin -p admin -t admin
2017-01-05 22:15:56 root [INFO] 2.begining evacuate_monitor!Pacemaker execute NovaEvacuate with Monitor option
2017-01-05 22:15:56 root [INFO] Current Pacemaker cluster has evacuation attribute as follow(attrd_updater -n evacuate -A):
Could not query value of evacuate: attribute does not exist
2017-01-05 22:15:57 root [INFO] Call handle_evacuatioms function to handle evacuation!
2017-01-05 22:15:57 root [INFO] 3.Begining handle evacuation:node is , current evacuation is ,and will entry into while loop!
2017-01-05 22:15:57 root [INFO] Now jump out the while loop!
2017-01-05 22:16:01 root [INFO] NovaEvacuate agnet stop successfull,directory /var/run/resource-agents/nova-evacuate.active removed!OCF state is 0
2017-01-05 22:16:08 root [INFO] 1.begining evacuate_validate!
2017-01-05 22:16:08 root [INFO] the fence_options is: -k http://192.168.142.203:35357/v2.0/ -l admin -p admin -t admin
2017-01-05 22:16:08 root [INFO] Start NovaEvacuate!directory /var/run/resource-agents/nova-evacuate.active will be created!
2017-01-05 22:16:09 root [INFO] 1.begining evacuate_validate!
2017-01-05 22:16:09 root [INFO] the fence_options is: -k http://192.168.142.203:35357/v2.0/ -l admin -p admin -t admin
2017-01-05 22:16:09 root [INFO] 2.begining evacuate_monitor!Pacemaker execute NovaEvacuate with Monitor option
2017-01-05 22:16:09 root [INFO] Current Pacemaker cluster has evacuation attribute as follow(attrd_updater -n evacuate -A):
Could not query value of evacuate: attribute does not exist
2017-01-05 22:16:09 root [INFO] Call handle_evacuatioms function to handle evacuation!
2017-01-05 22:16:09 root [INFO] 3.Begining handle evacuation:node is , current evacuation is ,and will entry into while loop!
2017-01-05 22:16:09 root [INFO] Now jump out the while loop!
2017-01-05 22:16:19 root [INFO] 1.begining evacuate_validate!
2017-01-05 22:16:19 root [INFO] the fence_options is: -k http://192.168.142.203:35357/v2.0/ -l admin -p admin -t admin
2017-01-05 22:16:19 root [INFO] 2.begining evacuate_monitor!Pacemaker execute NovaEvacuate with Monitor option
2017-01-05 22:16:19 root [INFO] Current Pacemaker cluster has evacuation attribute as follow(attrd_updater -n evacuate -A):
Could not query value of evacuate: attribute does not exist
2017-01-05 22:16:19 root [INFO] Call handle_evacuatioms function to handle evacuation!
2017-01-05 22:16:19 root [INFO] 3.Begining handle evacuation:node is , current evacuation is ,and will entry into while loop!
2017-01-05 22:16:19 root [INFO] Now jump out the while loop!
```

图 13-2 nova-evacuate 资源重启后 NovaEvacuate 重定向日志输出

2. RA 程序手动执行

另外一种调试 RA 程序的方式就是通过 sh 或 bash 命令直接执行 RA 脚本, 但是根据 13.1.2 节的介绍, RA 程序是不能在命令行环境中直接执行的, 在执行之前必须为其创建所需的环境变量。在 Pacemaker 集群中, 要手工执行 RA 程序, 可按如下步骤实现 (这里以 Nova-Evacuate 资源代理和 nova-evacuate 资源为例):

使 RA 相关的资源脱离 Pacemaker 的管理, 如下:

```
pcsresource unmanage nova-evacuate
```

配置 RA 运行所需的环境变量:

```
export OCF_ROOT=/usr/lib/ocf
```

```
export OCF_RESKEY_auth_url="http://192.168.142.203:35357/v2.0/"
```

```
export OCF_RESKEY_username="admin"
export OCF_RESKEY_password="admin"
export OCF_RESKEY_tenant_name="admin"
```

通过 RA 支持的 Action 参数运行脚本，如下：

```
bash/usr/lib/ocf/resource.d/openstack/NovaEvacuate start; echo $?
```

执行上述命令后，查找有用的错误信息，并查看返回代码。如果上述输出不能满足跟踪需要，则可以通过 `sh -x` 或 `bash -x` 来执行 RA 脚本，从而可以清晰地观察 RA 脚本执行过程中每一步进行了哪些操作，以及变量值的变化情况，如下：

```
bash/usr/lib/ocf/resource.d/openstack/NovaEvacuate stop
bash -x /usr/lib/ocf/resource.d/openstack/NovaEvacuate start; echo $?
```

手工调试执行完成后，记得将资源恢复交由 Pacemaker 管理，如下：

```
pcsresource manage nova-evacuate
```

上述过程可以通过一个完整的 shell 脚本来实现，例如自定义一个名为 `resource-debug.sh` 的脚本，在需要调试 `nova-evacuate` 所对应的 `NovaEvacuate` 资源代理时，只需将资源名称 (`nova-evacuate`) 和所需的操作 (如 `start`) 传递给此脚本，脚本会自动获取与资源对应的 RA 及其存放位置，自动提取资源配置参数并设置 RA 运行所需的环境变量，资源调试脚本的使用过程如下：

```
bash resource-debug.sh nova-evacuate start
```

`resource-debug.sh` 的脚本具有通用性，可以对任何 OCF 类型资源进行调试，脚本实现过程参考如下：

```
#!/bin/bash
//要调试的资源名称和操作分别赋值给resource和operation变量
resource=$1
operation=$2
//先使资源脱离Pacemaker的管理
pcs resource unmanage $resource
//设置OCF_ROOT环境变量
export OCF_ROOT=/usr/lib/ocf
//设置传递给RA的参数变量(参数由资源定义中获取)，并将其导出到当前环境中
eval $(pcs resource show $resource|grep Attributes:|awk '{s1=""; print}'|tr " " "\n"|\
grep .|xargs -I X echo export OCF_RESKEY_X|sed 's/=(.*)/="\1"/')
//RA的参数环境变量已经准备就绪，可以通过bash -x运行RA
bash -x /usr/lib/ocf/resource.d/$(pcs resource show $resource|sed -n 1p|\
sed 's@.*provider=(.*) type=(.*)@1/2@') $operation
```

`resource-debug.sh` 执行完成后，记得将资源恢复给 Pacemaker 管理，如之前对 `nova-evacuate` 资源进行了调试，则重新将其交由 Pacemaker 管理，如下：

```
pcs resource manage nova-evacuate
```

13.2 Pacemaker 集群调试与管理维护

13.2.1 Pacemaker 集群日志系统设置

Pacemaker 日志是观察和跟踪 Pacemaker 集群各个进程操作行为的关键，Pacemaker 集群节点和资源的启动、停止以及资源在不同状态之间的转换均会输出到 Pacemaker 日志中，因此 Pacemaker 日志设置是后续进行 Pacemaker 资源相关故障调试的基础。在以前的 Pacemaker 版本中，日志默认输出到系统 syslog 中，并且仅将 NOTICE 以上级别的概要日志输出到 syslog 中，而在 Pacemaker 1.1.-13 版本中，日志默认输出到 `/var/log/cluster/corosync.log` 文件中，由于 Pacemaker 日志不再与其他系统进程日志共同输出到 syslog 的 `/var/log/messages` 中，而是独立输出到自己专属的日志文件中，因此极大方便了 Pacemaker 日志的跟踪观察。运行在 Corosync 上层的 Pacemaker 继承了 Corosync 的日志设置方式，即通过 `/etc/corosync/corosync.conf` 文件进行配置，而运行在 CMAN 上层的 Pacemaker 则主要通过 `/etc/cluster/cluster.conf` 进行配置。在本书介绍的基于 Pacemaker 的高可用 OpenStack 集群中，Pacemaker 运行在 Corosync 顶层（Corosync 作为 Pacemaker 的依赖程序而被自动安装），因此本节将重点介绍如何在 `/etc/corosync/corosync.conf` 中配置 Pacemaker 日志。

Pacemaker 集群中，Corosync 进程主要提供集群范围内可靠的节点通信机制，同时负责实时同步节点之间的集群配置信息，此外，Corosync 还负责确认节点是否可以成为集群成员并在集群 Quorum 机制得到保证或失去后向 Pacemaker 发出通知。整体而言，Corosync 提供了集群内部的消息层并管理着系统和资源的可用性。而在 Pacemaker 集群中，节点 `corosync.conf` 配置文件保存了控制 Corosync 进程运行方式和执行过程的参数，而这些参数也在很大程度上控制了 Pacemaker 集群的正常运行。在同一个 Pacemaker 集中，当进行集群创建或节点加入授权时，`corosync.conf` 文件被传递至全部集群节点中，即所有 Pacemaker 集群节点具有相同的 `corosync.conf` 文件，并由 Corosync 进程实时保持该文件的一致性。在本书第 11 章中创建的 Pacemaker 集群中，OpenStack 三个控制节点组成了 Pacemaker 集群，每个控制节点上 `corosync.conf` 配置文件的内容如下：

```
[root@controller2-vm ~]# more /etc/corosync/corosync.conf
totem {
  version: 2
  secauth: off
    cluster_name: openstack-ha
  transport: udpu
}
nodelist {
  node {
    ring0_addr: controller1-vm
  }
  nodeid: 1
}
node {
```

```

        ring0_addr: controller2-vm
nodeid: 2
    }
node {
    ring0_addr: controller3-vm
nodeid: 3
    }
}

quorum {
provider: corosync_votequorum
}

logging {
    to_logfile: yes
logfile: /var/log/cluster/corosync.log
    to_syslog: yes
}

```

这里需要注意的是，尽管最终的 Pacemaker 集群由三个集群节点（OpenStack 控制节点）和两个远程节点（OpenStack 计算节点）构成，但是在 corosync.conf 文件中，节点列表部分仅有三个控制节点的定义，而没有远程节点的定义，这是因为运行 Pacemaker_remote 的远程节点并没有运行 Corosync 进程，因此也不需要 corosync.conf 文件，而该文件中自然也不会有远程节点的定义。corosync.conf 由不同的配置项构成，如 totem、nodelist、quorum 和 logging，其中 nodelist 部分定义了 Corosync 进行通信的 Ring，为了保证节点通信的高可用性，每个 Corosync 集群中可以有多个 Ring（此处仅有一个 Ring，即 Ring0），每个 Ring 通过节点 IP 地址将全部集群节点“串联”构成一个通信环，如果每个节点均配置有多个网段的管理 IP 地址，则一个 Corosync 集群中可以存在多个通信环，从而保证任一个 Ring 因网络故障而失效后，集群仍然可以保证有冗余的 Ring 而不会出现脑裂情况（Brain-Split），通过如下方式可以查看每个节点的 Ring 状态信息：

```

[root@controller2-vm ~]# corosync-cfgtool -s
Printing ring status.
Local node ID 2
RING ID 0
id      = 192.168.142.111
status  = ring 0 active with no faults

```

在 corosync.conf 配置文件中，除了 nodelist，还有一个重要的配置项 logging。而 logging 正是控制 Pacemaker 日志行为的配置项，在 logging 的配置项中，不同的配置参数控制了 Pacemaker 日志的不同输出形式，logging 中的配置参数解释如下：

- ❑ **Timestamp**：设置在所有日志消息中置入的时间戳，默认值为 off。
- ❑ **Fileline**：指定在日志消息中应该输出的文件名称和行号，默认值为 off。
- ❑ **Function_name**：指定在日志消息中应该输出的代码函数名称，默认值为 off。
- ❑ **to_stderr**：将日志消息定向到标准错误输出，可能的值为 yes 或 no。

- ❑ `to_syslog`: 将日志消息输出到系统 `syslog`, 可能的值为 `yes` 或 `no`。
- ❑ `to_logfile`: 将日志消息输出到自定义的日志文件, 可能的值为 `yes` 或 `no`。
- ❑ `logfile`, 如果 `to_logfile` 被设置为 `yes`, 则该选项指定自定义日志文件的路径。
- ❑ `logfile_priority`, 日志文件记录日志的优先级别, 如果 `debug` 被置为 `on` 则忽略此选项, 可能的值包括 `alert`、`crit`、`debug`、`emerg`、`err`、`info`、`notice` 和 `warning`, 默认值为 `info`。
- ❑ `syslog_facility`: 当 `to_syslog` 被设置为 `yes` 时, 该选项指定使用的系统 `syslog` 设备类型, 可能的值包括 `daemon`、`local0`、`local1`、`local2`、`local3`、`local4`、`local5`、`local6` 和 `local7`, 默认值是 `daemon`。
- ❑ `syslog_priority`: `syslog` 记录日志的优先级, 如果 `debug` 被置为 `on` 则忽略此选项, 可能的值包括 `alert`、`crit`、`debug`、`emerg`、`err`、`info`、`notice` 和 `warning`, 默认值为 `info`。
- ❑ `debug`: 设置是否在日志中输出 `debug` 信息, 默认值是 `off`。

通常情况下, 如果没有特别的需求, 使用 Pacemaker 默认的 `corosync.conf` 文件配置参数即可得到大多数所需的日志信息, 在 Pacemaker1-1-13 中, 默认的 Pacemaker 日志文件为 `/var/log/cluster/corosync.log`, 在使用默认的 `corosync.conf` 文件时, OpenStack 高可用集群中的 Pacemaker 日志输出如图 13-3 所示。

```
[root@controller3-va ~]# tail -f /var/log/cluster/corosync.log
apache@horizon[10799]: 2017/01/07 15:33:22 INFO: Successfully retrieved http header at http://192.168.142.112:80
Delay[cellmaster-delay][10232]: 2017/01/07 15:33:26 INFO: Delay is running OK
Jan 07 15:33:30 [1471] controller3-va cib: info: cra_compress_string: Compressed 289596 bytes into 15769 (ratio 18:1) in 39ms
Jan 07 15:33:31 [1471] controller3-va cib: info: cra_compress_string: Compressed 289599 bytes into 15782 (ratio 18:1) in 36ms
Jan 07 15:33:31 [1471] controller3-va cib: info: cra_compress_string: Compressed 289599 bytes into 15776 (ratio 18:1) in 36ms
apache@horizon[11389]: 2017/01/07 15:33:32 INFO: Successfully retrieved http header at http://192.168.142.112:80
apache@horizon[12010]: 2017/01/07 15:33:42 INFO: Successfully retrieved http header at http://192.168.142.112:80
lrmd: 2017/01/07 15:33:45 INFO: rabbitmq-cluster: get_monitor(): CHECK LEVEL 13: 0
lrmd: 2017/01/07 15:33:46 INFO: rabbitmq-cluster: get_monitor(): get_status() returns 0.
lrmd: 2017/01/07 15:33:46 INFO: rabbitmq-cluster: get_monitor(): also checking if we are master.
Delay[cellmaster-delay][11659]: 2017/01/07 15:33:46 INFO: Delay is running OK
lrmd: 2017/01/07 15:33:46 INFO: rabbitmq-cluster: get_monitor(): master attribute is 0
lrmd: 2017/01/07 15:33:46 INFO: rabbitmq-cluster: get_monitor(): checking if rabbit app is running
lrmd: 2017/01/07 15:33:47 INFO: rabbitmq-cluster: get_monitor(): rabbit app is running, checking if we are the part of healthy cluster
lrmd: 2017/01/07 15:33:47 INFO: rabbitmq-cluster: get_monitor(): rabbit app is running, looking for master on controller3-va
lrmd: 2017/01/07 15:33:47 INFO: rabbitmq-cluster: get_monitor(): fetched master attribute for controller3-va. attr value is 0
lrmd: 2017/01/07 15:33:47 INFO: rabbitmq-cluster: get_monitor(): rabbit app is running, master is controller3-va
lrmd: 2017/01/07 15:33:48 INFO: rabbitmq-cluster: get_monitor(): rabbit app is running and is member of healthy cluster
lrmd: 2017/01/07 15:33:48 INFO: rabbitmq-cluster: get_monitor(): preparing to update master score for node
lrmd: 2017/01/07 15:33:48 INFO: rabbitmq-cluster: get_monitor(): comparing our uptime (8922) with controller1-va (7875)
lrmd: 2017/01/07 15:33:48 INFO: rabbitmq-cluster: get_monitor(): comparing our uptime (8922) with controller2-va (0)
lrmd: 2017/01/07 15:33:48 INFO: rabbitmq-cluster: get_monitor(): we are the oldest node
lrmd: 2017/01/07 15:33:49 INFO: rabbitmq-cluster: su_rabbit_cmd(): the invoked command exited 0: /usr/sbin/rabbitmqctl list_channels 2>&1 > /dev/null
lrmd: 2017/01/07 15:33:49 INFO: rabbitmq-cluster: su_rabbit_cmd(): the invoked command exited 0: /usr/sbin/rabbitmqctl -q eval 'rabbit_alarm.get_alarm
lrmd: 2017/01/07 15:33:50 INFO: rabbitmq-cluster: su_rabbit_cmd(): the invoked command exited 0: /usr/sbin/rabbitmqctl -q list_queues memory messages
lrmd: 2017/01/07 15:33:52 INFO: rabbitmq-cluster: su_rabbit_cmd(): the invoked command exited 0: /usr/sbin/rabbitmqctl -q status
lrmd: 2017/01/07 15:33:52 INFO: rabbitmq-cluster: get_monitor(): RabbitMQ is running 244 queues consuming 8.03905m of 4823m total, with 402 queued mes
lrmd: 2017/01/07 15:33:52 INFO: rabbitmq-cluster: get_monitor(): RabbitMQ status: [ipid,5153], [running_applications,[rabbit,'RabbitMQ','3.8.0'], [os
lrmd: 2017/01/07 15:33:52 INFO: rabbitmq-cluster: get_monitor(): get_monitor function ready to return 5
```

图 13-3 Pacemaker 日志输出

13.2.2 Pacemaker 集群日志构成分析

在 RHEL7 或 CentOS7 等 Linux 系统中, Pacemaker 默认将日志定向输出到单一日志文件中 (`/var/log/cluster/corosync.log`), 在该日志文件中, 可以发现集群引擎 `pacemakerd` 及其他组件的日志信息, 如 `crmd`、`lrmd` 和 `cib` 等。对于 Pacemaker 集群中的每个操作, 大量的日志被输出到该文件中, 在集群正常运行时, 管理员很少会注意日志文件, 但是在集群故障诊断排除时, 由于 Linux 集群系统每秒钟都有大量进程在执行, 并不断输出日志信息到日志文件中, 因此依靠简单的资源监视命令很难在日志文件中追踪故障进程的操作历史。

此时，如果清楚构成集群日志文件信息的各个部分，在故障诊断时便可针对性地分析日志，另外，写入日志文件信息中的错误代码通常已进行了预定义，同时根据资源类型的不同还可以看到某些执行过程中的命令输出，因此在 Pacemaker 集群故障问题分析时，首先需要定位该故障与 Pacemaker 集群的哪个进程组件相关，之后再从日志文件中提取与该进程相关的全部信息，并根据信息中的错误代码和可能的针对某个资源的执行命令追溯集群进程对资源的操作历史，从而定位问题的故障点并进一步解决问题。

在 Pacemaker 默认的日志文件 /var/log/cluster/corosync.log 中，每个日志条目均由不同的信息段组成，每个信息段之间以空格分开。其中，前三列表示日志输出的时间戳，第四列表示输出日志的进程号，第五列表示主机名称，第六列表示进程名称，这里的进程名称与第四列的进程号是对应的，第七列表示日志的类型（info、notice 等）和执行日志输出的预定义函数名称，其余部分为函数内部预定义的输出文本信息，Pacemaker 日志条目信息构成如图 13-4 所示，其中输出日志的进程名称和预定义消息文本是故障诊断过程中最重要的两个部分。

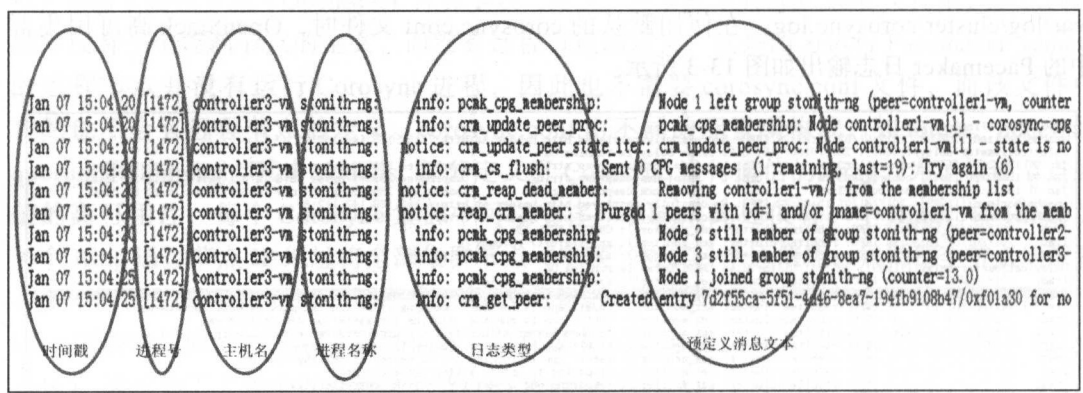


图 13-4 Pacemaker 日志条目构成信息

在 Pacemaker 故障诊断过程中，熟悉 Pacemaker 不同进程所负责的主要功能是故障分析时在日志文件中定位对应进程消息的基础，如与节点隔离相关的故障问题通常与 Stonithd 进程相关，与资源相关的故障通常与 crmd 或 lrmd 进程相关。下面对 Pacemaker 集群中最常用的进程及其输出日志进行简单介绍，更多信息请参考 Pacemaker 官方网站 <http://clusterlabs.org/>。

1. Stonith-ng

Stonith 进程负责对集群节点进行隔离（Fencing），在 Pacemaker 集群中可以配置不同类型的 Fencing 设备，当存在以下几种情况时，将会接收到 Stonith 进程发出的日志信息：

- ❑ Stonith 拓扑出现任何变化，如新节点或故障恢复节点的加入；
- ❑ Stonith 接收到 Pacemaker 发出的 Fence 命令；
- ❑ Fence 命令的状态输出。

Stonith 进程默认将日志信息输出到 Pacemaker 日志文件中，因此要查看 Stonith 进程相关的日志信息，可以通过如下 grep 命令实现：

```
grep stonith-ng /var/log/cluster/corosync.log
```

Stonith 进程输出的日志信息如图 13-5 所示。

```
[root@controller3-va ~]# grep -i stonith-ng /var/log/cluster/corosync.log
Jan 07 13:09:49 [1472] controller3-va stonith-ng: info: pcak_cpg_membership: Node 2 left group stonith-ng (peer=controller2-va, counter=3.0)
Jan 07 13:09:49 [1472] controller3-va stonith-ng: info: cra_update_peer_proc: pcak_cpg_membership: Node controller2-va[2] - corosync-cpg is
Jan 07 13:09:49 [1472] controller3-va stonith-ng: notice: cra_update_peer_state_iter: cra_update_peer_proc: Node controller2-va[2] - state is now lo
Jan 07 13:09:49 [1472] controller3-va stonith-ng: info: cra_cs_flush: Sent 0 CPG messages (1 remaining, last=4): Try again (6)
Jan 07 13:09:49 [1472] controller3-va stonith-ng: notice: cra_reap_dead_member: Removing controller2-va/2 from the membership list
Jan 07 13:09:49 [1472] controller3-va stonith-ng: notice: reap_cra_member: Purged 1 peers with id=2 and/or unname=controller2-va from the membersh
Jan 07 13:09:49 [1472] controller3-va stonith-ng: info: pcak_cpg_membership: Node 1 still member of group stonith-ng (peer=controller1-va,
Jan 07 13:09:49 [1472] controller3-va stonith-ng: info: pcak_cpg_membership: Node 3 still member of group stonith-ng (peer=controller3-va,
Jan 07 13:10:02 [1472] controller3-va stonith-ng: info: pcak_cpg_membership: Node 2 joined group stonith-ng (counter=4.0)
Jan 07 13:10:02 [1472] controller3-va stonith-ng: info: pcak_cpg_membership: Node 1 still member of group stonith-ng (peer=controller1-va,
Jan 07 13:10:02 [1472] controller3-va stonith-ng: info: cra_get_peer: Created entry d5d65787-4db8-4789-b81b-47b36fe54558/0xe6ff40 for node c
```

图 13-5 Stonith 进程输出日志信息示例

2. Corosync

Corosync 是 Pacemaker 高可用集群的重要组成部分，Corosync 向集群引擎提供了完整的基础架构功能，这些基础架构功能包括通信机制、集群成员机制和心跳机制。此外，Corosync 与 Pacemaker 集群中很多进程均有交互，Corosync 也会将日志信息输出到 Pacemaker 日志文件中，与其他进程的日志输出不一样，Corosync 输出的日志中，各个 Corosync 守护进程输出的消息会以守护进程名称进行标记，如 MAIN、QUORUM 和 CPG 等。当存在以下几种情况时，将会接收到 Corosync 发出的日志信息：

- 以 TOTEM 命名的 Heartbeat 协议出现任何错误或配置变更；
- 以 MAIN 命名的 corosync 守护进程相关的信息；
- 以 QUORUM 命名的集群 Quorum 状态变化信息；
- 以 CPG 命名的成员关系更新或改变；
- 集群管理器，如以 SERV 和 PCMK 命名的 pacemaker 服务级别的更改和调试信息。

与 Corosync 相关的日志信息如图 13-6 所示。

```
[1022] controller2-va corosyncnotice [TOTEM] A new membership (192.168.142.110:18512) was formed. Members
[1022] controller2-va corosyncnotice [QUORUM] Members[3]: 1 2 3
[1022] controller2-va corosyncnotice [MAIN] Completed service synchronization, ready to provide service.
[1022] controller2-va corosyncwarning [MAIN] Corosync main process was not scheduled for 1519.3464 ms (threshold is 1320.0000 ms).
[1022] controller2-va corosyncnotice [TOTEM] A processor failed, forming new configuration.
[1022] controller2-va corosyncnotice [TOTEM] A new membership (192.168.142.110:18516) was formed. Members
[1022] controller2-va corosyncnotice [QUORUM] Members[3]: 1 2 3
[1022] controller2-va corosyncnotice [MAIN] Completed service synchronization, ready to provide service.
[1022] controller2-va corosyncnotice [TOTEM] A processor failed, forming new configuration.
[1022] controller2-va corosyncnotice [TOTEM] A new membership (192.168.142.110:18520) was formed. Members
[1022] controller2-va corosyncnotice [QUORUM] Members[3]: 1 2 3
[1022] controller2-va corosyncnotice [MAIN] Completed service synchronization, ready to provide service.
```

图 13-6 Corosync 相关的日志信息

3. CRM Daemon

crmd 是 Pacemaker 中主要的集群管理进程，通常可以在 crmd 输出的日志信息中查询到很多有用的信息，当集群正常运行时，crmd 进程在后台进行很多操作，因此 crmd 输出的日志信息非常频繁。在使用 crmd 日志进行故障诊断时，通过日志可以定位集群故障出现的地方，以及为了解决问题集群管理器做了哪些操作。crmd 输出日志主要记录了以下相关信息：

- ❑ cib.conf 文件更新；
- ❑ 资源当前状态和资源状态改变；
- ❑ 诸如 start、stop 和 monitor 等资源操作；
- ❑ 任何一个资源出现故障；
- ❑ Stonith 进行隔离操作。

在 Pacemaker 日志文件中，通过以下 grep 命令可以提取 crmd 相关的日志信息：

```
grep -i crmd /var/log/cluster/corosync.log
```

与 crmd 进程相关的日志信息如图 13-7 所示。

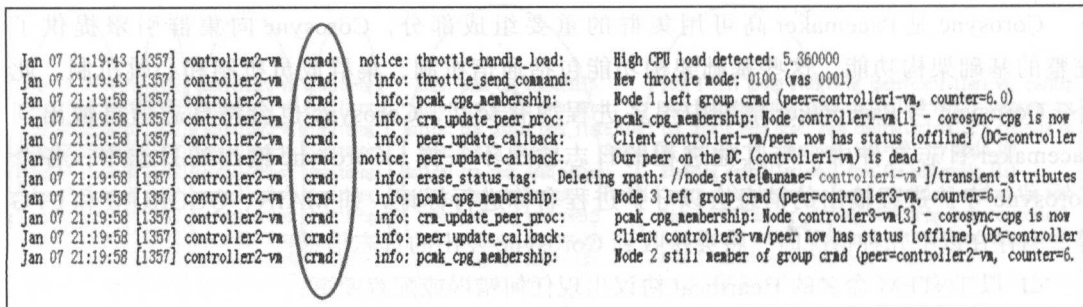


图 13-7 crmd 进程相关的日志信息

4. CIB

CIB 是集群配置文件，cib.conf 是记录全部集群配置信息的 XML 格式文件，cib 也会向 Pacemaker 集群日志文件输出信息，与 cib 相关的日志信息主要包括以下几个部分：

- ❑ Quorum 更新；
- ❑ 服务启动期间 CIB 信息校验失败；
- ❑ CIB 读写错误。

在 Pacemaker 日志文件中，通过以下 grep 命令可以提取 cib 相关的日志信息：

```
grep -i cib /var/log/cluster/corosync.log
```

与 cib 进程相关的日志信息如图 13-8 所示。

5. Policy Engine

策略引擎 (Policy Engine, PE) 定义了集群资源管理器 (CRM) 的行为操作，换句话说，

Pacemaker 针对集群资源的全部操作都必须“有据可依”，而 PE 定义了 CRM 的操作路线图。PE 进程输出的日志主要与以下信息相关：

```
Jan 07 15:01:58 [2100] controller2-vm cib: info: cib_perform_op: Diff: --- 0.1925.995 2
Jan 07 15:01:58 [2100] controller2-vm cib: info: cib_perform_op: Diff: +++ 0.1925.996 (null)
Jan 07 15:01:58 [2100] controller2-vm cib: info: cib_perform_op: + /cib: @num_updates=996
Jan 07 15:01:58 [2100] controller2-vm cib: info: cib_perform_op: + /cib/status/node_state[@id='3']: @cra-debug-origin=post_cache_update
Jan 07 15:01:58 [2100] controller2-vm cib: info: cib_perform_op: + /cib/status/node_state[@id='1']: @cra-debug-origin=post_cache_update
Jan 07 15:01:58 [2100] controller2-vm cib: info: cib_perform_op: + /cib/status/node_state[@id='computer1']: @cra-debug-origin=post_cache_update
Jan 07 15:01:58 [2100] controller2-vm cib: info: cib_perform_op: + /cib/status/node_state[@id='computer2']: @cra-debug-origin=post_cache_update
Jan 07 15:01:58 [2100] controller2-vm cib: info: cib_perform_op: + /cib/status/node_state[@id='2']: @cra-debug-origin=post_cache_update
Jan 07 15:01:58 [2100] controller2-vm cib: info: cib_process_request: Completed cib_modify operation for section status: OK (rc=0, origin=controller1-
Jan 07 15:02:04 [2100] controller2-vm cib: info: cib_process_ping: Reporting our current digest to controller1-vm: 7809bc28343c68d41fda206fab57626 for 0.1
Jan 07 15:02:14 [2100] controller2-vm cib: info: cra_compress_string: Compressed 284459 bytes into 15401 (ratio 18:1) in 2525ms
Jan 07 15:02:26 [2100] controller2-vm cib: info: cib_perform_op: Diff: --- 0.1925.996 2
Jan 07 15:02:26 [2100] controller2-vm cib: info: cib_perform_op: Diff: +++ 0.1925.997 (null)
Jan 07 15:02:26 [2100] controller2-vm cib: info: cib_perform_op: + /cib: @num_updates=997
Jan 07 15:02:26 [2100] controller2-vm cib: info: cib_perform_op: + /cib/status/node_state[@id='1']: @cra-debug-origin=do_update_resource
Jan 07 15:02:26 [2100] controller2-vm cib: info: cib_perform_op: ++ /cib/status/node_state[@id='1']/lra[@id='1']/lra_resources/lra_resource[@id='heat-api
Jan 07 15:02:26 [2100] controller2-vm cib: info: cib_perform_op:
```

图 13-8 CIB 相关的日志信息

- CRM 开始使用策略，如资源失败后，CRM 将根据策略进行重启操作；
- 与 PE 图形文件（Graph File）相关的信息，集群配置中全部资源之间的依赖和相互关系通过图形化的方式呈现，无论何时资源状态出现变化，都会生成新的图形文件，而 PE 进程将与此相关的信息记录到日志文件中。

在 Pacemaker 日志文件中，通过以下 grep 命令可以提取 Policy Engine 相关的日志信息：

```
grep -i pengine /var/log/cluster/corosync.log
```

与 PE 进程相关的日志信息如图 13-9 所示。

```
Jan 07 13:41:23 [2154] controller1-va pengine: info: native_print: fence1 (stonith:fence_xvm): Started controller2-va
Jan 07 13:41:23 [2154] controller1-va pengine: info: native_print: fence2 (stonith:fence_xvm): Started controller1-va
Jan 07 13:41:23 [2154] controller1-va pengine: info: native_print: fence3 (stonith:fence_xvm): Started controller2-va
Jan 07 13:41:23 [2154] controller1-va pengine: info: clone_print: Clone Set: lb-haproxy-clone [lb-haproxy]
Jan 07 13:41:23 [2154] controller1-va pengine: info: short_print: Started: [ controller1-va controller2-va controller3-va ]
Jan 07 13:41:23 [2154] controller1-va pengine: info: short_print: Stopped: [ computer1 computer2 ]
Jan 07 13:41:23 [2154] controller1-va pengine: info: native_print: vip-db (ocf::heartbeat:IPaddr2): Started controller3-va
Jan 07 13:41:23 [2154] controller1-va pengine: info: native_print: vip-rabbitmq (ocf::heartbeat:IPaddr2): Started controller1-va
Jan 07 13:41:23 [2154] controller1-va pengine: info: native_print: vip-keystone (ocf::heartbeat:IPaddr2): Started controller2-va
Jan 07 13:41:23 [2154] controller1-va pengine: info: native_print: vip-glance (ocf::heartbeat:IPaddr2): Started controller3-va
Jan 07 13:41:23 [2154] controller1-va pengine: info: native_print: vip-cinder (ocf::heartbeat:IPaddr2): Started controller1-va
Jan 07 13:41:23 [2154] controller1-va pengine: info: native_print: vip-swift (ocf::heartbeat:IPaddr2): Started controller2-va
Jan 07 13:41:23 [2154] controller1-va pengine: info: native_print: vip-neutron (ocf::heartbeat:IPaddr2): Started controller3-va
Jan 07 13:41:23 [2154] controller1-va pengine: info: native_print: vip-neva (ocf::heartbeat:IPaddr2): Started controller1-va
Jan 07 13:41:23 [2154] controller1-va pengine: info: native_print: vip-horizon (ocf::heartbeat:IPaddr2): Started controller2-va
Jan 07 13:41:23 [2154] controller1-va pengine: info: native_print: vip-heat (ocf::heartbeat:IPaddr2): Started controller3-va
Jan 07 13:41:23 [2154] controller1-va pengine: info: native_print: vip-celometer (ocf::heartbeat:IPaddr2): Started controller1-va
Jan 07 13:41:23 [2154] controller1-va pengine: info: native_print: vip-oid (ocf::heartbeat:IPaddr2): Started controller2-va
Jan 07 13:41:23 [2154] controller1-va pengine: info: clone_print: Master/Slave Set: galera-master [galera]
Jan 07 13:41:23 [2154] controller1-va pengine: info: short_print: Masters: [ controller1-va controller2-va controller3-va ]
Jan 07 13:41:23 [2154] controller1-va pengine: info: short_print: Stopped: [ computer1 computer2 ]
```

图 13-9 Policy Engine 相关的日志信息

13.2.3 Pacemaker 集群日志调试分析

在 Pacemaker 集群中，故障诊断的第一步便是查看集群日志文件中是否有 ERROR 或 WARN 严重级别的日志信息，查看方式如下：

```
grep -e ERROR -e WARN /var/log/cluster/corosync.log
```

如果在 Pacemaker 日志文件中没有发现任何 ERROR 或 WARN 级别的日志信息，则进一步观察日志文件中来自 crmd 进程的信息，查看方式如下：

```
grep -e crmd\[ -e crmd: /var/log/cluster/corosync.log
```

如果未能看到来自 crmd 的任何日志信息，则参照 13.2.1 节检查 Pacemaker 集群日志配置是否正确，日志是否正确输出到 syslog 或特定日志文件中（这里为 /var/log/cluster/corosync.log）。此外，虽然 crmd 进程运行在全部集群节点上，但是诊断信息的提取仅依赖本地节点 crmd 进程。对于节点级别的故障，用以故障诊断的日志文件应该取自集群的 DC 节点，而对于资源级别的故障，诊断日志应该取自 DC 节点和资源故障的节点。在 crmd 日志中，如果出现如下所示的条目信息：

```
[1357] controller2-vm crmd: notice: crm_update_peer_state_iter: crm_reap_unseen_nodes: Node controller3-vm[3] - state is now lost (was member)
[1357] controller2-vm crmd: notice: crm_update_peer_state_iter: crm_reap_unseen_nodes: Node controller1-vm[1] - state is now lost (was member)
```

则表示对应的节点已经不再是集群成员节点，可能的原因是节点被关闭或出现故障。而如果 crmd 日志信息中出现如下条目：

```
[1357] controller2-vm crmd: notice: crm_update_peer_state_iter: pcmk_quorum_notification: Node controller3-vm[3] - state is now member (was lost)
[1357] controller2-vm crmd: notice: crm_update_peer_state_iter: pcmk_quorum_notification: Node controller1-vm[1] - state is now member (was lost)
```

则表明节点又重新加入到集群中。如果 crmd 日志信息中出现类似如下的条目：

```
crmd: notice: run_graph: Transition 16 (Complete=105, Skipped=3, Incomplete=20, Source=/var/lib/pacemaker/pengine/pe-input-1182.bz2): Stopped
crmd: info: do_state_transition: State transition S_TRANSITION_ENGINE -> S_POLICY_ENGINE [ input=I_PE_CALC cause=C_FSA_INTERNAL origin=notify_crmd ]
```

则表明集群正在尝试恢复。如果在 crmd 日志信息中出现以下条目：

```
controller3-vm crmd: notice: te_rsc_command: Initiating action 202: monitor glance-registry:l_monitor_60000 on controller1-vm
controller3-vm crmd: notice: te_rsc_command: Initiating action 209: start glance-api_start_0 on controller3-vm (local)
controller3-vm crmd: notice: te_rsc_command: Initiating action 211: start glance-api:l_start_0 on controller1-vm
controller3-vm crmd: notice: te_rsc_command: Initiating action 210: monitor glance-api_monitor_60000 on controller3-vm (local)
```

则表明集群正在执行资源操作，如 start、stop 和 monitor 等。上述 crmd 日志中，集群正在对 controller1-vm 上的 glance-registry 和 controller3-vm 上的 glance-api 资源进行健康检查 (monitor)，并且正在启动 (start) controller1-vm 和 controller3-vm 上的 glance-api 资源。如果 crmd 日志信息中存在以下条目：

```
[12155] controller1-vm crmd: notice: te_fence_node: Executing reboot fencing operation (549) on controller2-vm (timeout=60000)
```

则表明集群正在对节点进行隔离操作，上述输出信息中，集群正在对 controller2-vm 节点执行 reboot 类型的隔离操作。就 Pacemaker 集群而言，最常见的故障问题主要是节点级别的故障和资源级别的故障，下面针对不同级别的故障诊断方式进行简单介绍。

1. 节点级别故障

(1) 检查 crmd 进程是否记录到相关的故障

如果从 crmd 日志的 crm_update_peer_state() 中不能发现任何有用的条目消息，则检查 Pacemaker 的日志文件，确认集群节点成员信息是正确和最新的。

(2) 确认 crmd 是否能够启动恢复操作

如果从 crmd 日志的 `do_state_transition` 和 `run_graph()` 中不能发现有用的条目消息, 则说明集群对于节点故障没有做出任何反应, 此时需要参考 Pacemaker 日志设置的相关信息以获取更多关于 crmd 为何忽略节点故障的原因^①。

(3) 确认 crmd 是否能够执行恢复操作

如果在 crmd 日志的 `do_state_transition()` 中确实发现了条目信息, 但是 `run_graph()` 条目中包括了 “Complete=0, Pending=0, Fired=0, Skipped=0, Incomplete=0” 文本信息, 则表明集群认为当前并不需要做任何响应。此时, 需要获取 `run_graph()` 条目中给出的文件 (如 `/var/lib/pacemaker/pengine/pe-input-1182.bz2`), 之后参考 Policy Engine 相关的调试过程^②。

(4) 检查集群是否尝试隔离故障节点

首先确认集群的 `stonith-enabled` 属性是否被设置为 `true`, 如果已经设置, 则获取 `run_graph()` 条目中给出的文件 (如 `/var/lib/pacemaker/pengine/pe-input-1182.bz2`), 之后参考 Policy Engine 相关的调试过程。

(5) 隔离操作是否成功完成

如果隔离操作不能成功完成, 则需要检查 Fencing 资源的配置情况, 如果有必要, 则进入 Stonith 故障排除环节。

2. 资源级别故障

(1) 确认资源是否真的已经故障

如果资源本身仍在正常运行, 则检查与资源名称相匹配的日志信息, 以确认为何资源代理认为资源出现了故障。此外, 检查资源代理源代码, 以确认日志中的信息由源代码中的哪个异常语句抛出, 并跟踪该语句的输入输出情况。

(2) crmd 是否记录到资源故障

如果 crmd 没有记录到相应的资源故障事件, 则检查与资源名称相匹配的日志信息, 以核实是否资源代理输出了相关的故障信息。此外, 检查针对故障资源是否配置了循环监控操作。

(3) crmd 是否启动了资源恢复操作

如果从 crmd 日志的 `do_state_transition` 和 `run_graph()` 中不能发现有用的条目消息, 则说明集群对于资源故障没有做出任何反应, 此时需要参考 Pacemaker 日志设置的相关信息以获取更多关于 crmd 为何忽略资源故障的原因。

(4) crmd 是否成功执行恢复操作

如果在 crmd 日志的 `do_state_transition()` 中确实发现了条目信息, 但是 `run_graph()` 条目中包括了 “Complete=0, Pending=0, Fired=0, Skipped=0, Incomplete=0” 文本信息, 则表明集群认为当前并不需要做任何响应。此时, 需要获取 `run_graph()` 条目中给出的文件 (如

① <http://blog.clusterlabs.org/blog/2013/pacemaker-logging/>

② <http://blog.clusterlabs.org/blog/2013/debugging-pengine>

/var/lib/pacemaker/pengine/pe-input-1182.bz2), 之后参考 Policy Engine 相关的调试过程。

(5) 检查资源是否出现意外的 start/stop/move 或在正常的 start/stop/move 时出现失败

如果出现了上述情况, 则需要获取 run_graph() 条目中给出的文件 (如 /var/lib/pacemaker/pengine/pe-input-1182.bz2), 之后参考 Policy Engine 相关的调试过程。

13.2.4 Pacemaker 集群 GUI 管理界面

在基于 Pacemaker 的 OpenStack 高可用集群中, 除了 OpenStack 提供了高可用的 Dashboard 图形化用户管理界面, 集群资源管理器 Pacemaker 和负载均衡器 HAProxy 均提供了图形化管理界面 (GUI)。在本文介绍的 OpenStack 高可用集群部署方案中, Pacemaker 集群运行了二百多个资源, 而如果仅通过 pcs 命令行对这些资源进行管理, 势必增加运维管理员的负担, 并且复杂的 pcs 命令行参数也不便于理解记忆, 因此 Pacemaker 提供了图形化的人机交互界面, 用户通过 Pacemaker GUI 管理界面即可轻松实现对 Pacemaker 集群及其资源的管理操作。此外, HAProxy 负载均衡器也提供了 GUI 管理界面, 通过管理界面用户可以实时监控与 OpenStack 服务相关的访问请求。

1. Pacemaker 图形化用户管理界面

Pacemaker 提供了基于 Web 的接口界面用以集群管理, 并且通过单个 Web 界面即可管理多个集群。默认情况下, Pacemaker 图形管理界面使用 2224 端口, 因此通过 https://nodename:2224 即可访问 Pacemaker 图形化管理界面。在 Pacemaker 的图形化管理界面中, 用户可以进行以下操作:

- ☐ 创建一个新的集群;
- ☐ 添加已经存在的集群到 GUI 中;
- ☐ 管理集群节点, 如对节点进行 start、stop 和 standby 操作;
- ☐ 配置集群 Fence 设备;
- ☐ 配置集群资源;
- ☐ 资源属性设置, 如 order、location、colocation 等;
- ☐ 集群属性设置;
- ☐ 角色创建。

通常情况下, 要访问 Pacemaker 的 GUI 界面无须做其他设置, pcsd 进程负责 Pacemaker 的 GUI 服务, 通过如下命令可以查看 PCS GUI 服务是否已经启动:

```
[root@controller3-vm ~]# systemctl status pcsd
```

pcsd 进程默认使用 hacluster 用户提供服务, 而在集群初始化时, 通常需要对 hacluster 用户进行密码设置, 本文中已为 hacluster 用户设置了密码 (密码为 hacluster)。现在, 通过 Pacemaker 集群任一节点的 IP 地址即可访问 GUI 界面, 如图 13-10 所示。在图 13-10 中, 用户名输入 hacluster, 密码输入 hacluster 即可进入 Pacemaker GUI 的管理界面。

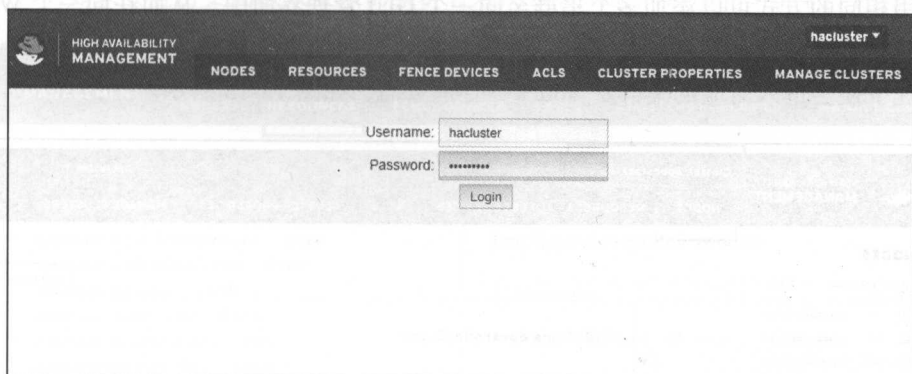


图 13-10 Pacemaker 集群 GUI 登录界面

默认情况下，Pacemaker GUI 界面中不存在任何集群，如图 13-11 所示。因此需要用户自己新建或者添加已存在的集群至 GUI 中。

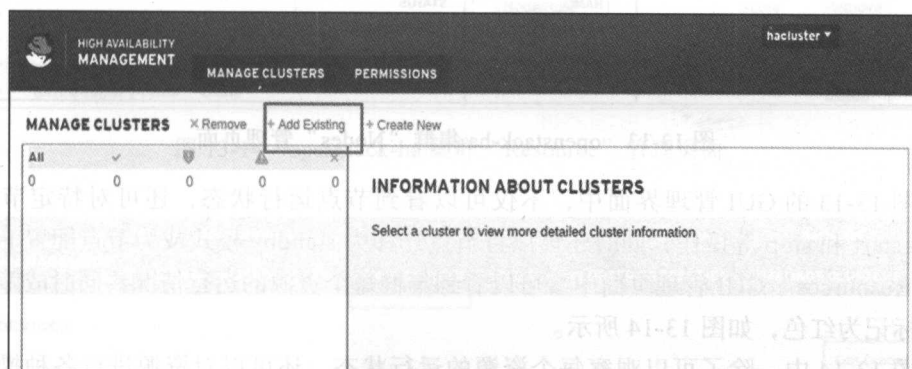


图 13-11 初次登录的 Pacemaker GUI 界面

添加集群至 GUI 的方式很简单，在如图 13-12 中，输入任一集群节点的 IP 地址，之后点击“Add Existing”，集群添加过程会自动将集群信息推送至 GUI 界面中。

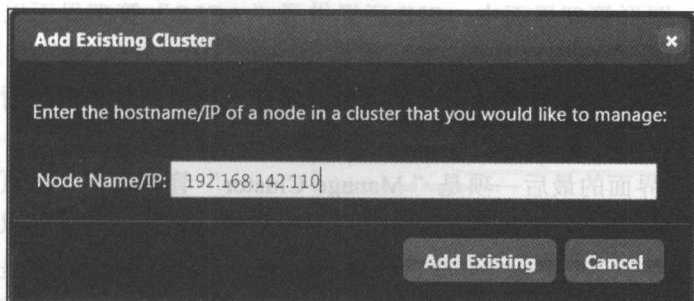


图 13-12 添加集群至 GUI 界面中

使用相同的方式可以添加多个集群至同一个 GUI 管理界面中，从而在同一个 Web 站点上即可管理多个集群。集群添加完成后，选择想要管理的集群名称（这里为 *openstack-ha*），默认情况下便会进入对应的集群“Nodes”管理页面，如图 13-13 所示。

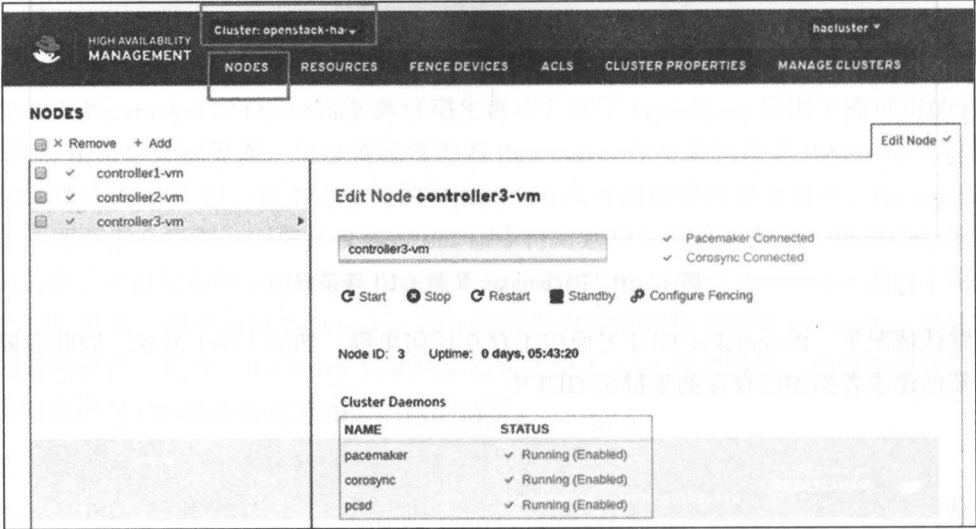


图 13-13 openstack-ha 集群“Nodes”管理页面

在图 13-13 的 GUI 管理界面中，不仅可以看到节点运行状态，还可对特定节点进行 start、restart 和 stop 等操作，同时还可以将节点迁移为 standby 模式或为节点配置 Fencing。而在“Resources”GUI 管理页面中，可以看到集群每个资源的运行情况，同时故障资源会自动被标记为红色，如图 13-14 所示。

在图 13-14 中，除了可以观察每个资源的运行状态，还可以对资源进行各种操作，如添加 Order、Location 和 Colocation 约束，以及添加元数据属性和对资源进行 enable、disable 和 cleanup 等操作。在“Fence Device”GUI 管理页面中，可以看到集群全部的 Fence 设备及其运行状态，还可对其进行 Remove 和 Cleanup 等操作，同时还可通过“Add”添加新的 Fence 设备，如图 13-15 所示。

在 Pacemaker 图形管理界面中，GUI 还提供了“ACLS”管理界面以供用户进行访问控制列表设置，但是此功能在 Pacemaker 集群中很少使用。“ACLS”的右侧是“Cluster Properties”管理页面，如图 13-16 所示，在集群属性管理页面中，可以看到集群的全部属性参数及其属性值，并可在该页面中更新并应用特定的集群属性。

集群管理 GUI 界面的最后一项是“Manage Cluster”管理页面，该页面即是最初登录 GUI 时呈现的初始界面，在此界面中可以进行创建集群、添加已有集群以及删除集群等针对集群的操作。整体而言，Pacemaker 提供的 GUI 管理界面比较直观地呈现了集群运行状态，并以菜单的形式提供了部分集群管理功能，不过相对强大的 pcs 命令行管理工具，GUI 并未覆盖全部的集群管理功能。因此，如果仅对集群进行常规的管理设置，则 GUI 是不错的

管理工具，而如果要进行更高级或复杂的集群管理配置，则应该首选 pcs 命令行管理工具。

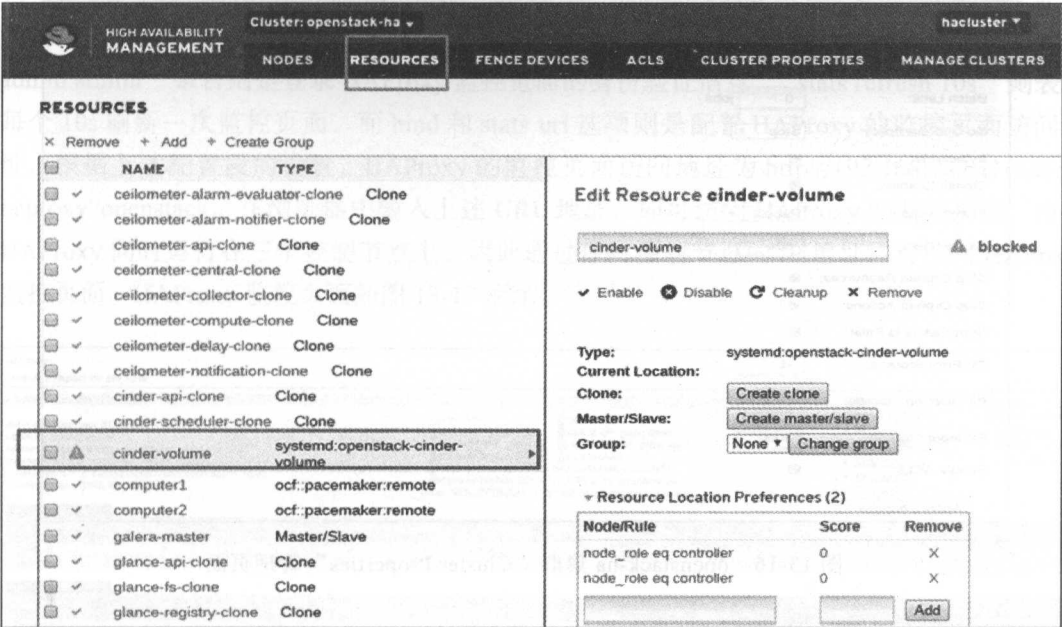


图 13-14 openstack-ha 集群 “Resource” 管理页面

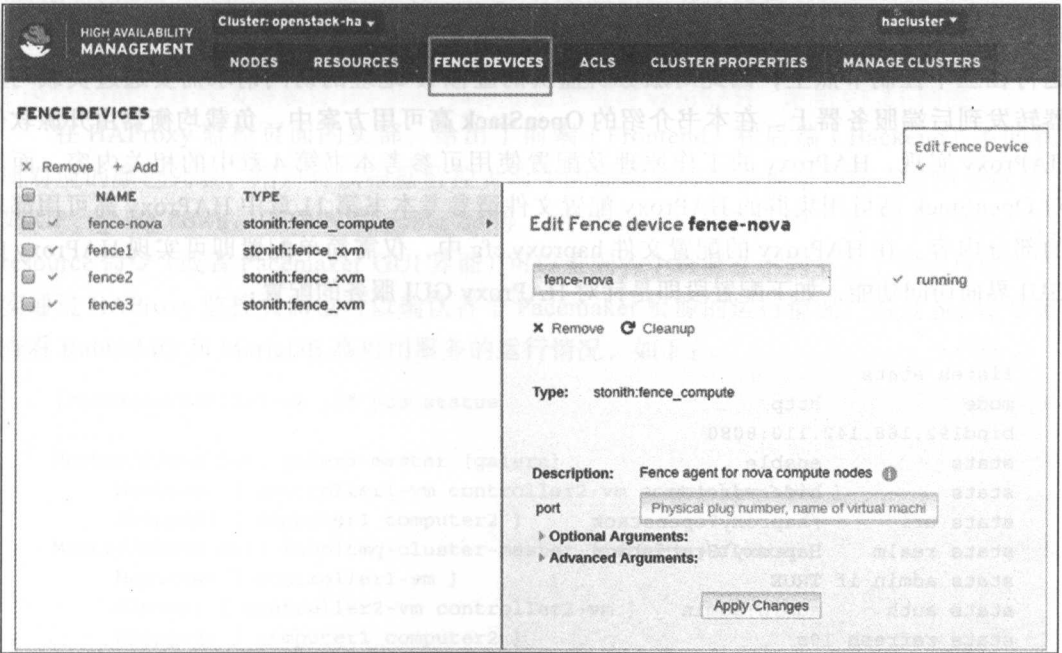


图 13-15 openstack-ha 集群 “Fence Device” 管理页面

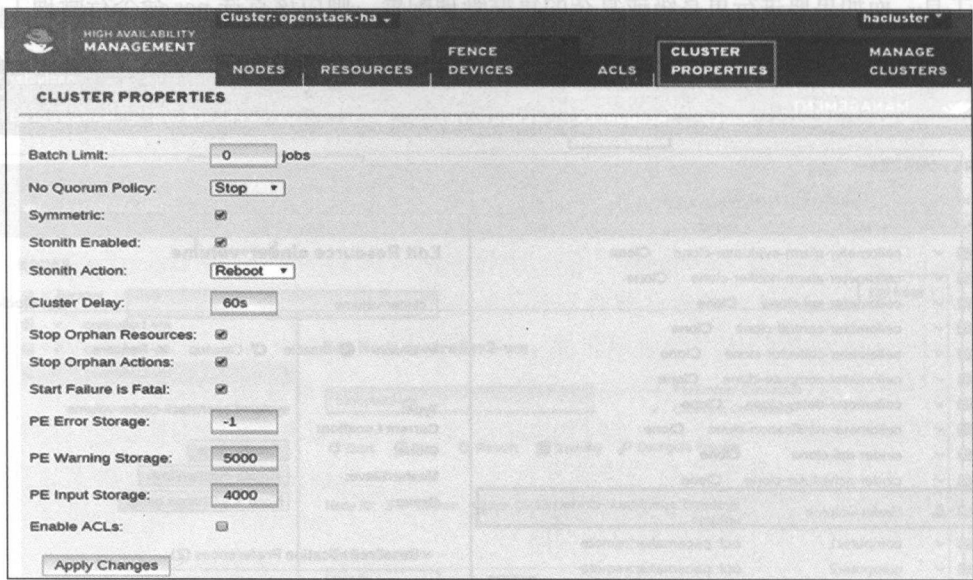


图 13-16 openstack-ha 集群 “Cluster Properties” 管理页面

2. HAProxy 图形化用户管理界面

在 OpenStack 高可用集群配置部署中，为了实现 OpenStack 控制服务的高可用性，为每一项 OpenStack 服务提供了专门的虚拟 IP 地址，外部客户端或 OpenStack 内部组件通过虚拟 IP 地址访问目标服务，由于高可用集群中的各项服务均以 A/P 或 A/A 高可用模式运行在三个控制节点上，因此对服务所监听的虚拟 IP 地址的访问请求需要通过负载均衡器转发到后端服务器上。在本书介绍的 OpenStack 高可用方案中，负载均衡器由开源软件 HAProxy 实现，HAProxy 的工作原理及配置使用可参考本书第 4 章中的相关内容，而针对 OpenStack 高可用集群的 HAProxy 配置文件请参考本书第 11 章中 HAProxy 高可用部署的部分内容。在 HAProxy 的配置文件 haproxy.cfg 中，仅需简单配置即可实现 HAProxy 的 GUI 界面访问功能，如下配置段即是针对 HAProxy GUI 服务的配置：

```
.....
listen stats
mode          http
bind192.168.142.110:8080
stats         enable
stats         hide-version
stats uri     /haproxy?openstack
stats realm   Haproxy\Statistics
stats admin if TRUE
stats auth    admin:admin
stats refresh 10s
.....
```


由于 HAProxy 以 Clone 资源形式以 A/A 高可用模式运行在三个控制节点上，因此三个控制节点上 haproxy.cfg 的内容应该完全一致（各个节点 HAProxy 的监听 IP 地址不同）。在上述配置段中，“stats enable” 条目即是启用 HAProxy 的监控页面，“stats auth admin:admin” 条目则是登录 HAProxy 监控页面的身份验证信息，“stats refresh 10s” 则表示每个 10s 刷新一次监控页面，而 bind 和 stats url 选项则是配置 HAProxy 的监控页面访问地址，根据上述配置段的设置，HAProxy 的监控页面访问地址为 <http://192.168.142.110:8080/haproxy?openstack>。在浏览器中输入上述 URL 地址，即可访问 HAProxy 的监控页面，由于 HAProxy 同时运行在三个控制节点上，因此通过任一控制节点的 IP 地址均可访问 HAProxy 监控页面。HAProxy 监控页面如图 13-17 所示。

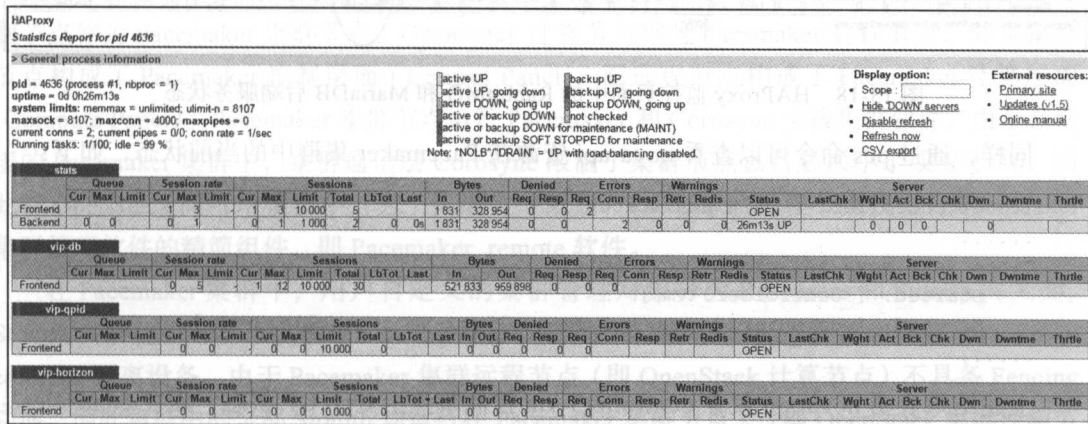


图 13-17 HAProxy 监控页面

在 HAProxy 监控页面的头部，给出了前端（Frontend）和后端（Backend）不同状态所对应的颜色列表，用户可以根据监控页面中的前端和后端颜色来判断当前所处的状态，如 UP、going down、DOWN 和 going up 等。在 Pacemaker 集群中，通过 pcs stats 或 pcs resource 命令（或者 Pacemaker GUI 界面）可以看到各个资源在不同节点上的运行状态，其实通过 HAProxy 监控页面也可以确认各个 Pacemaker 资源的运行情况。通过 pcs 命令可以查看 RabbitMQ 和 MariaDB 高可用服务的运行情况，如下：

```
[root@controller1-vm ~]# pcs status
.....
Master/Slave Set: galera-master [galera]
Masters: [ controller1-vm controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
Master/Slave Set: rabbitmq-cluster-master [rabbitmq-cluster]
Masters: [ controller1-vm ]
Slaves: [ controller2-vm controller3-vm ]
Stopped: [ computer1 computer2 ]
.....
```


上述输出表明 RabbitMQ 和 MariaDB 的 Master/Slave 多状态资源在三个控制节点上均正常运行，这一结果在 HAProxy 监控页面上可以得到验证，如图 13-18 所示，对应于 RabbitMQ 和 MariaDB 的 HAProxy 后端在三个控制节点上均处于 UP 状态。

db-vms-galera																												
Queue		Session rate		Sessions							Bytes		Denied		Errors		Warnings		Status	LastChk	Server							
				Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis			Wght	Act	Bck	Chk	Dwn	Downtime	Thrtle	
controller1-vm		0	0	0	0	5	47	47	113	2	324	1344	133	2249	160	0	0	0	0	0m11s UP	* L7OK/200 in 772ms	1	-	Y	7	2	3m14s	-
controller2-vm		0	0	0	0	3	18	18	1	7m44s	15276	24142	0	0	0	0	0	0	0	4m3s UP	* L7OK/200 in 416ms	1	-	Y	14	5	5m5s	-
controller3-vm		0	0	0	0	0	0	0	0	0	7	0	0	0	0	0	0	0	0	4m9s UP	* L7OK/200 in 13ms	1	-	Y	16	6	3m31s	-
Backend		0	0	0	0	5	47	47	1000	131	3	324	1369	409	2273	302	0	0	0	0	37m52s UP		1	0	3	2	2m58s	-
Choose the action to perform on the checked servers: <input type="button" value="Apply"/>																												

rabbitmq-vms																												
Queue		Session rate		Sessions							Bytes		Denied		Errors		Warnings		Status	LastChk	Server							
				Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis			Wght	Act	Bck	Chk	Dwn	Downtime	Thrtle	
controller1-vm		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	7m36s UP	L4OK in 59ms	1	Y	-	10	3	21m55s	-
controller2-vm		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4m27s UP	L4OK in 195ms	1	Y	-	11	3	21m52s	-
controller3-vm		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	14m9s UP	L4OK in 0ms	1	Y	-	1	1	26m51s	-
Backend		0	0	0	0	0	0	0	1000	0	0	0	0	0	0	0	0	0	0	18m7s UP		3	3	0	2	17m21s	-	
Choose the action to perform on the checked servers: <input type="button" value="Apply"/>																												

图 13-18 HAProxy 监控页面中的 RabbitMQ 和 MariaDB 后端服务状态

同样，通过 pcs 命令可以查看 nova-api 资源在 Pacemaker 集群中的当前状态，如下：

```
[root@controller1-vm ~]# pcs status
.....
Clone Set: nova-api-clone [nova-api]
  Started: [ controller3-vm ]
  Stopped: [ computer1 computer2 controller1-vm controller2-vm ]
.....
```

上述结果表明 nova-api 仅在 controller3-vm 控制节点上处于 UP 状态，而在 controller2-vm 和 controller1-vm 上均处于 DOWN 状态，同样的结果在 HAProxy 监控页面中也可以得到验证，如图 13-19 所示，HAProxy 监控页面显示与 nova-api 资源相关的 HAProxy 后端服务仅在 controller3-vm 上处于 UP 状态，而在 controller2-vm 和 controller1-vm 后端节点上均为 DOWN 状态。

vip nova-api																											
Queue		Session rate		Sessions					Bytes		Denied		Errors		Warnings		Status	Server									
				Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp		Retr	Redis	LastChk	Wght	Act	Bck	Chk	Dwn	Downtime	Thrtle
Frontend		0	0	0	0	0	10000	0	0	0	0	0	0	0	0	0	0	OPEN									

nova-api-vms																											
Queue		Session rate		Sessions					Bytes		Denied		Errors		Warnings		Status	Server									
				Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp		Retr	Redis	LastChk	Wght	Act	Bck	Chk	Dwn	Downtime	Thrtle
controller1-vm		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5m5s DOWN	L4CON in 7ms	1	Y	-	18	5	32m50s	-	
controller2-vm		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1m17s DOWN	L4CON in 4ms	1	Y	-	20	2	25m36s	-	
controller3-vm		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15m8s UP	L4OK in 0ms	1	Y	-	1	1	31m15s	-	
Backend		0	0	0	0	0	0	1000	0	0	0	0	0	0	0	0	0	19m34s UP		1	1	0	2	24m7s	-		
Choose the action to perform on the checked servers: <input type="text"/> <input type="button" value="Apply"/>																											

图 13-19 HAProxy 监控页面中的 Nova-api 后端服务状态

HAProxy 的监控页面不仅可以直观显示各个 Pacemaker 资源的运行状态，还会对与各个前 / 后端服务相关的连接数目、会话数目、错误次数和等待队列等参数信息进行统计，关于 HAProxy 监控页面的参数理解可以参考第 4 章的相关内容。整体而言，通过 HAProxy 提

供的 GUI 监控页面，用户不仅可以对各个前 / 后端服务的运行状态进行直观掌握，还可以对整个 OpenStack 高可用集群的网络连接和服务进行全局性的监控。

13.3 OpenStack 实例高可用原理分析与问题诊断

13.3.1 OpenStack 高可用集群计算节点资源配置

在基于 Pacemaker 的 OpenStack 高可用集群中，计算节点运行 Pacemaker_remote 并以 Pacemaker 集群远程节点形式加入 Pacemaker 集群中，从而使得计算节点运行的 OpenStack 相关服务由控制节点组成的 Pacemaker 集群统一控制。在 Pacemaker 集群中，OpenStack 控制节点称为 Pacemaker 集群节点，OpenStack 计算节点称为 Pacemaker 远程节点，并且集群节点构成了 Pacemaker 控制层面（Control Panel），而远程节点构成了 Pacemaker 计算层面（Compute Panel）。Pacemaker 集群节点运行 Pacemaker 和 Corosync 全栈集群软件，由于在常规的 Pacemaker 集群中，集群通信层 Corosync 限制了集群节点数目必须小于或等于 16，因此在 OpenStack 高可用集群中，为了突破这一限制，Pacemaker 远程节点仅运行 Pacemaker 集群管理软件的精简组件，即 Pacemaker_remote 软件。

在 Pacemaker 集群中，用户自定义的集群管理对象分为 Resources 和 Stonith 两大类，Stonith 主要包括计算节点物理 Fencing 设备、控制节点物理 Fencing 设备和计算节点 Nova-compute 隔离设备。由于 Pacemaker 集群远程节点（即 OpenStack 计算节点）不具备 Fencing 功能，因此集群中的全部 Stonith 均运行在 Pacemaker 集群节点上（即 OpenStack 集群控制节点上），在本书介绍的 OpenStack 高可用集群方案实验环境中，Pacemaker 集群中的 Stonith 如下所示：

```
[root@controller1-vm]#pcs stonith
fence1 (stonith:fence_xvm):      Started controller3-vm
fence2 (stonith:fence_xvm):      Started controller1-vm
fence3 (stonith:fence_xvm):      Started controller2-vm
fence-nova (stonith:fence_compute): Started controller3-vm
```

由于实验环境中计算节点没有 Fencing 设备，因此采用手工 Fencing 形式模拟计算节点的隔离，而在生产环境中，Pacemaker 集群中对应每个计算节点必须存在一个基于物理服务器 Fencing 设备（如 IPMI）的 Stonith。节点 Fencing 是 Pacemaker 集群中极为关键的环节，对故障节点的 Fencing 操作是保证集群服务正确运行和集群数据安全完整性最可靠的措施，因此在生产环境中，全部 Pacemaker 集群的成员节点都必须要求其具备 Fencing 设备并能被 Pacemaker 隔离。除了 Stonith，Pacemaker 集群中用户自定义最多的管理对象便是集群资源 Resources，Resource 是 Pacemaker 管理调度的最小单元，从理论上讲，任何可以在 Linux 系统中运行的服务均可以在 Pacemaker 集群中以资源形式运行，并接受 Pacemaker 的管理调度。在基于 Pacemaker 的 OpenStack 高可用环境中，每一项 OpenStack 及其依赖的服务

都被定义为资源，资源在 Pacemaker 集群中以 A/A 或 A/P 高可用模式运行从而保证服务的高可用性。在本书介绍的 OpenStack 高可用集群中，Pacemaker 集群最终运行的全部资源可参见 12.3.1 节。

在 Pacemaker 集群资源中，通过资源的 Location 约束和节点的属性设置，可以将不同的资源运行在不同的节点集中，如将 OpenStack API 服务限制在控制节点运行，而将 Nova Compute 服务限制在计算节点运行。例如要将 Nova-evacuate 资源限制在具有 node_role=controller 属性的节点集（控制节点中）范围内运行，则其 Location 约束的设置如下：

```
pcs constraint location nova-evacuate rule \
resource-discovery=exclusivescore=0 node_role eq controller --force
```

对于大规模的 OpenStack 集群，资源发现方式选项 resource-discovery 对 Pacemaker 集群运行性能有很大影响，该选项默认值为 always（总是发现），即 Pacemaker 总是对被约束到该节点上的资源进行 Discovery 操作。而这里 resource-discovery 选项值被设置为 exclusive（独占发现），其目的就在于告诉 Pacemaker，nova-evacuate 资源仅能运行在控制节点上。当可预测集群会有较大规模节点扩展时，将资源发现操作仅限制到具备运行资源条件的节点子集时，对于集群资源启动和运行性能均会有极大提高，尤其当 Pacemaker_remote 用于达到 100 节点规模的集群时，此选项对群集运行速度可能产生巨大影响^①。由于 Pacemaker 集群逻辑上被划分为控制层面和计算层面，因此在基于 Pacemaker 的 OpenStack 高可用集群中，每一个资源都要进行带有 exclusive 参数的 Location 约束设置，从而将每个资源限制到仅在控制层面或计算层面节点集上运行。

13.3.2 OpenStack 高可用集群 Fence_compute 分析

Fence_compute 是一个用于 OpenStack 计算节点实例自动恢复的 Fence Agent，由 Fence-agents-compute 安装包提供，其已被集成在 RedHat 发行的 RHEL7 或开源 Centos7 版本 Linux 系统中，用户可以直接对其进行安装使用。在 OpenStack 计算节点实例高可用部署中，Fence_compute 是个极为关键的代理程序，其主要作用在于告知 Pacemaker 集群 Nova 计算节点已处于 down 状态并重新将故障计算节点上的实例调度至其他正常节点。本质上 Fence_compute 是由 Python 语言编写的脚本^②，主要用于同 Pacemaker 集群和 OpenStack API 进行交互，如为 Pacemaker 集群节点设置 evacuate 属性、获取计算节点 Nova 服务状态、设置计算节点 Nova 服务状态和调用 Nova API 进行实例 Evacuate 迁移等操作。作为一个命令行工具（/usr/sbin/fence_compute），Fence_compute 既可以接受命令行参数，也可以接受 stdin 标准输入参数，当作为 Pacemaker 资源 Agent 运行时，其接受 stdin 输入参数，而作为普通命令行调用时，接受命令行输入参数。在功能测试和命令行调用过程中，fence_

① http://hopperman24.rssing.com/chan-13011538/all_p5.html

② https://github.com/ClusterLabs/fence-agents/blob/master/fence/agents/compute/fence_compute.py

compute 最常用的命令行参数及其参数功能解释如下:

```
-n, --plug=[id]                //节点主机名称
-S, --password-script=[script] //存放password的文件
-k, --auth-url=[url]           //Keystone Admin授权URL
-p, --password=[password]      //Keystone登录访问密码
-e, --endpoint-type=[endpoint] //Nova Endpoint类型, 默认为internalURL
-t, --tenant-name=[tenant]     //Keystone Admin租户名称
-o, --action=[action]          //隔离操作, 默认为off
-l, --username=[name]          //Keystone登录用户名
-d, --domain=[string]          //-n参数指定主机所在的Domain名称
--instance-filtering           //允许实例Evacuate, 默认为True
--no-shared-storage            //禁用与共享存储相关的功能, 默认为False, 仅在非共享存储环境中使用
--record-only                  //仅将目标主机记录为需要撤离状态, 不进行实际的撤离操作, 默认为False
-D, --debug-file=[debugfile]  //将Debug信息写入指定文件中
```

对应每个命令行输入参数名称, Fence_compute 均有对应的 stdin 标准输入参数名称, 与命令行输入参数对应的标准输入参数如下:

```
port                //节点主机名称
passwd_script        //密码存放文件
auth-url            //Keystone Admin授权URL
passwd              //Keystone登录访问密码
endpoint-type       // Nova Endpoint类型, 默认为internalURL
tenant-name         //Keystone Admin租户名称
action              //隔离操作, 默认为off
login               //Keystone登录用户名
domain              //主机DNS domain
instance-filtering  //允许实例进行撤离操作, 默认为True
no-shared-storage   //禁用共享存储功能, 默认为False
record-only         //仅将目标记录为撤离状态而不进行撤离操作, 默认为False
debug               //将Debug信息写入指定文件中
```

在实际调用 Fence_compute 时, 需要告知 Fence_compute 代理要进行什么操作, 因此 Fence_compute 预定义了与节点相关的 Action, Fence_compute 将根据用户指定的命令行参数或标准输入参数, 以及指定的 Action 进行具体的操作。Fence_compute 可以进行的 Action 如下:

```
on                //对指定节点进行Power on操作
off               //对指定节点进行Power off操作
status           //返回指定计算节点的状态
list             //列出全部计算节点状态信息
list-status      //列出全部计算节点名称及其Power状态
monitor          //检查Fence设备的健康情况
metadata         //显示资源的XML元数据信息
```

在基于 Pacemaker/Pacemaker_remote 的 OpenStack 计算节点实例高可用配置中, Pacemaker 将以标准参数输入形式调用 Fence_compute, 该步骤发生在创建基于 Fence_compute 代理的 fence-nova 隔离设备时, 如下:

```
pcs stonith create fence-nova fence_computeauth-url=\
http://$vip_keystone:35357/v2.0/ login=admin passwd=admin\
tenant-name=admin record-only=1 --force
```

此时，Fence_compute 将以 Pacemaker 资源代理的形式由 Pacemaker 周期性调用，此处 在创建 fence-nova 时，以标准参数输入形式向 Fence_compute 传入了与 Keystone 相关的认 证信息，此外需要注意的是“record-only=1”选项，由于 Fence_compute 对 record-only 参 数的默认值为 False，因此在创建 fence-nova 时必须显式指定 record-only 的选项值为 True， 以告知 Fence_compute 仅记录下需要进行 Evacuate 的目标主机即可。在另外的 Pacemaker 资源代理 NovaEvacuate 脚本中，Fence_compute 将被再次调用，由于 NovaEvacuate 是 shell 编写的脚本，因此 Fence_compute 在 NovaEvacuate 中将以命令行参数形式被调用，如下：

```
fence_compute -o off auth_url=http://$vip_keystone:35357 \
username=admin password=admin tenant_name=admin -n compute1
```

NovaEvacuate 仅在监测到 Pacemaker 集群中有节点的 evacuate 属性值为“yes”时才会 调用 Fence_compute，并且调用时会传入计算节点名称（-n compute1）和 Action（-o off）参 数，同时使用默认的 record-only=False 参数，而 Fence_compute 在接收到上述参数后，将 调用 Nova 的 evacuate API 对指定计算节点上的实例进行 Evacuate 操作。在用 Python 编写 的 Fence_compute Fence 代理程序中，调用 Nova API 进行实例 Evacuate 操作的源代码段如 下所示：

```
.....
//定义实例迁移函数
def _server_evacuate(server, on_shared_storage):
    logging.debug("(Entry into _server_evacuate)Now begining evacuate instances on %s"
        % server)
    success = True
    error_message = ""
    try:
        //调用Nova evacuate API进行实例迁移
        nova.servers.evacuate(server=server['uuid'], on_shared_storage=on_shared_storage)
    except Exception as e:
        success = False
        error_message = "Error while evacuating instance: %s" % e

    return {
        "server_uuid": server['uuid'],
        "evacuate_accepted": success,
        "error_message": error_message,
    }
.....
```

除了对计算节点实例进行 Evacuate 操作，Fence_compute 代理程序的另一主要功能在 于实现 Pacemaker 集群中计算节点 evacuate 属性的设置，当调用 Fence_compute 的命令行 参数为 record-only 设置了非 False 的选项值时，Fence_compute 将仅执行计算节点 evacuate 属

性值设置，而不调用 Nova 的 evacuate API 进行实例迁移。在 Fence_compute 的 Python 程序中，执行是否进行节点 evacuate 属性设置和对节点进行 evacuate 属性设置的源代码段如下：

```
.....
//计算节点evacuate属性设置函数。该函数通过shell命令行调用attrd_updater将指
//定计算节点(host参数值)的evacuate属性值设为“no”或“yes”(status参数值)
def set_attrd_status(host, status, options):
    logging.debug("Setting fencing status for %s to %s" % (host, status))
    run_command(options, "attrd_updater -p -n evacute -Q -N %s -v %s" % (host, status))
.....

//Python入口main()函数
def main():
    .....
    //如果record-only不为False，则仅设置节点的evacuate属性值。在创建fence-nova
    //时指定了record-only=1，因此fence-nova会执行以下代码段从而调用
    //set_attrd_status()函数将指定的计算节点evacuate属性值设为“no”或“yes”
    if options["--record-only"] != "False":
        //如果action为on，则将节点evacuate设置为“no”
        if options["--action"] == "on":
            set_attrd_status(options["--plug"], "no", options)
        sys.exit(0)
        //如果action为off/reboot，则将节点evacuate设置为“yes”
    elif options["--action"] in ["off", "reboot"]:
        set_attrd_status(options["--plug"], "yes", options)
        sys.exit(0)
    .....
```

在 Pacemaker 集群中，有个很重要的集群属性设置命令行函数 attrd_updater，该函数由 Pacemaker 程序安装包提供。Fence_compute 对计算节点 evacuate 属性设置和 NovaEvacuate 对集群计算节点 evacuate 属性值的实时监测都通过 attrd_updater 函数完成。attrd_updater 通常以命令行参数形式调用，常见的命令行参数如下：

```
-n, --name=value    //要设置的属性名称
-U, --update=value  //更新attrd中的属性值，兼容旧版本的-v参数
-Q, --query         //从attrd中查询属性值
-D, --delete        //删除attrd中的属性
-N, --node=value    //需要设置属性的节点名称，不指定则默认为本地节点
-A, --all           //形式所有集群节点的属性值
-p, --private       //如果创建的是一个新的集群属性，则不将此属性写入集群CIB中
```

在基于 Pacemaker 的 OpenStack 计算节点高可用配置中，fence-nova 以 Action 为 Status 或 Monitor 的操作调用 fence_compute 以实时监控集群中是否有计算节点 Nova 服务处于 down 状态，默认情况下 Pacemaker 每隔 60s 通过 fence-nova 执行一次健康检查，如果发现节点故障则以 -n、-N、-U/-v 和 -p 参数调用 attrd_updater 函数将该节点的 evacuate 属性设置为“yes”，而 nova-evacuate 则实时通过 -A 参数调用 attrd_updater 查询集群中是否存在 evacuate 属性值为“yes”的节点，如果发现且为计算节点则以 Action 为 off 的参数调用

fence_compute 进行实例迁移。当所有节点均正常运行时，通过 attrd_updater 命令行参数并不能查询到任何节点的 evacuate 属性，如下：

```
[root@controller1-vm ~]# attrd_updater -n evacuate -A
Could not query value of evacuate: attribute does not exist
```

attrd_updater 除了查询集群节点状态，还可以为集群中任意节点设置任意属性值，如下命令将为控制节点 controller1-vm 创建 evacuate 属性并将其值设置为 “yes”：

```
[root@controller1-vm ~]# attrd_updater -n evacuate -N controller1-vm \
-v yes
[root@controller1-vm ~]# attrd_updater -n evacuate -A
name="evacuate" host="controller1-vm" value="yes"
```

在实际应用中，为控制节点设置 evacuate 属性并无实际意义，这个操作并不会触发 NovaEvacuate 进行具体的操作，因为 NovaEvacuate 会自动过滤非计算节点主机，如图 13-20 所示，在 NovaEvacuate 的输出日志中，尽管 controller1-vm 有 evacuate 属性且其值为 “yes”，但是 Nova 已将控制节点 controller1-vm 过滤。

```
[INFO] 1.begining evacuate_validate!
[INFO] the fence_options is: -k http://192.168.142.203:35357/v2.0/-l admin -p admin -t admin
[INFO] 2.begining evacuate_monitor!Pacemaker execute NovaEvacuate with Monitor option
[INFO] Current Pacemaker cluster has evacuation attribute as follow(attrd_updater -n evacuate -A):
      name="evacuate" host="controller1-vm" value="yes"
[INFO] Call handle_evacuations function to handle evacuation!
[INFO] 3.Begining handle evacuation:node is controller1-vm,current evacuation is yes,and will entry into while loop!
[INFO] The state of controller1-vm evacuation is YES,controller1-vm need evacuate!
[INFO] Initiating evacuation of controller1-vm
[INFO] Nova does not know about controller1-vm
```

图 13-20 NovaEvacuate 过滤控制节点主机日志

NovaEvacuate 过滤主机的过程很简单，只需通过 fence_compute 的 list 操作查询 Pacemaker 集群中的全部计算节点列表，如果 NovaEvacuate 监控到的节点主机名不在此列表中，则将此主机过滤。例如，在本书介绍的三控制节点和两计算节点的 OpenStack 高可用集群中，fence_compute 查询到的计算节点如下：

```
[root@controller1-vm ~]# fence_compute -o list -k \
http://192.168.142.203:35357/v2.0/ -l admin -p admin -t admin
Get hypervisors list,if there is domain option
Starting new HTTP connection (1): 192.168.142.203
"POST /v2.0/tokens HTTP/1.1" 200 3867
Starting new HTTP connection (1): 192.168.142.210
"GET /v2/5751ec6ea581413484cad0bed9c39bf4/os-hypervisors/detail HTTP/1.1" 200
2617
computer1,
computer2,
```

fence_compute 经过 Keystone 认证后，调用 Nova 的 Hypervisors 查询 API，最终得到

的结果即 OpenStack 中的计算节点主机名称。通过 `fence_compute` 命令行中的不同 Action 参数，还可以查询到 OpenStack 计算节点的其他信息，如需要查看计算节点 `computer1` 当前的运行状态，通过 `status` 参数即可实现：

```
[root@controller1-vm ~]# fence_compute -o status -n computer1 -k \
http://192.168.142.203:35357/v2.0/ -l admin -p admin -t admin
Status: ON
```

在 OpenStack 实例高可用设计中，计算节点的故障监控、隔离和恢复是实例 HA 实现的要点，在基于 Pacemaker/pacemaker_remote 的实例 HA 方案中，节点隔离由 Pacemaker 利用 Fencing 设备自动完成，而监控和恢复则由 Fence_compute 和 NovaEvacuate 代理程序实现，本节介绍了由 Fence-agents-compute 提供的 fence_compute 代理程序在实例 HA 中的作用，下一节将重点分析 NovaEvacuate 资源代理程序。

13.3.3 OpenStack 高可用集群 NovaEvacuate 分析

NovaEvacuate 是 OCF 类型的资源代理程序，在 RedHat 发行的 RHEL7 或开源 Centos7 版本 Linux 系统中，NovaEvacuate 资源代理由 Resource-agents 软件安装包提供，其脚本文件位于操作系统的 `/usr/lib/ocf/resource.d/openstack` 目录中。NovaEvacuate 的主要作用在于实时跟踪监控 OpenStack 集群中是否有计算节点需要撤离并对被标记为需要撤离的计算节点进行实例迁移操作。NovaEvacuate 是一个标准的 OCF 类型 RA，在基于 Pacemaker 的 OpenStack 高可用部署中，通常基于此 RA 创建名为 `nova-evacuate` 的资源，`nova-evacuate` 资源仅运行在控制层面节点上，其创建过程如下：

```
pcs resource create nova-evacuate ocf:openstack:NovaEvacuate \
auth_url=http://$vip_keystone:35357 username=admin password=admin \
tenant_name=admin --force
```

`nova-evacuate` 资源必须在虚拟 IP 地址资源、Glance-api 资源、Nova-conductor 资源以及 Nova-compute 资源启动之后才能启动。对 `nova-evacuate` 资源而言，最重要的两个 Action 便是 `start` 和 `monitor`，而在 `start` 和 `monitor` 之前，NovaEvacuate 会默认调用 `evacuate_validate` 函数进行参数配置验证，如 Keystone 授权 URL、租户名称、用户名和用户密码等。NovaEvacuate 资源代理对应不同 Action 参数所执行的函数如下：

```
.....
case $__OCF_ACTION in
start)      evacuate_validate; evacuate_start;;
stop)       evacuate_stop;;
monitor)    evacuate_validate; evacuate_monitor;;
meta-data)  meta_data
            exit $OCF_SUCCESS
;;
usage|help) evacuate_usage
            exit $OCF_SUCCESS
```

```

;;
validate-all) exit $OCF_SUCCESS;;
*)
    evacuate_usage
    exit $OCF_ERR_UNIMPLEMENTED
;;
esac
.....

```

在 NovaEvacuate 定义的函数中, `evacuate_validate` 和 `evacuate_monitor` 是最重要的两个, 其中 `evacuate_validate` 负责参数验证并构造调用 `fence_compute` 所需的命令行参数, 而 `evacuate_monitor` 则负责监控 Pacemaker 集群中是否有计算节点需要进行撤离操作, 如果发现则对其上的实例进行迁移操作, `evacuate_monitor` 函数实现过程如下:

```

.....
evacuate_monitor() {
if [ ! -f "$statefile" ]; then
    return $OCF_NOT_RUNNING
fi
//跟踪节点evacuate属性值并调用handle_evacuatiions函数进行后续处理
    handle_evacuatiions $(attrd_updater -n evacute -A | tr '=' ' ' | awk '{print
        $4" "$6}')
return $OCF_SUCCESS
}
.....

```

`evacuate_monitor` 函数的功能便是通过 `attrd_updater` 的 `-A` 参数实时监控 Pacemaker 集群中的节点 `evacuate` 属性值, 并将监控到的节点主机名和其对应的 `evacuate` 属性值传递给 `handle_evacuation` 函数进行处理。`handle_evacuation` 主要负责判断该节点 `evacuate` 属性值是否为“yes”, 如果为“yes”则认为节点需要进行 Evacuate 操作, 之后再对 `evacuate` 值为“yes”的节点进行过滤以排除可能的非计算节点主机, 最后调用 `fence_compute` 进行计算节点实例迁移, 并根据迁移结果重置计算节点的 `evacuate` 属性值。`handle_evacuation` 函数实现过程如下:

```

handle_evacuatiions() {
while [ $# -gt 0 ]; do
    node=$1
    state=$2
    shift; shift;
    need_evacuate=0
//判断节点evacuate属性值情况, 可能的值为“yes”、“no”和空字符
    case $state in
        "" ) ;;
        no) ocf_log debug "$node is either fine or already handled";;
        yes) need_evacuate=1;;
        *@*)
            where=$(echo $state | awk -F@ '{print $1}')
            when=$(echo $state | awk -F@ '{print $2}')
            now=$(date +%s)

```

```

if [ (($now - $when)) -gt 60 ]; then
ocf_log info "Processing partial evacuation of $node by $where at $when"
need_evacuate=1
else
ocf_log info "Deferring processing partial evacuation of $node by
$where at $when"
fi
;;

esac

//如果之前的判断认为节点需要进行Evacuate, 则进行下一步操作
if [ $need_evacuate = 1 ]; then
found=0
ocf_log notice "Initiating evacuation of $node"
//判断Nova中是否存在此节点, 即仅对计算节点进行Evacuate操作
for known in $(fence_compute ${fence_options} -o list | tr -d ','); do
if [ ${known} = ${node} ]; then
found=1
break
fi
done
if [ $found = 0 ]; then
ocf_log info "Nova does not know about ${node}"
continue
fi
//调用update_evacuation函数更新节点的evacuate属性值
update_evacuation ${node} "$(uname -n)@$(date +%s)"
if [ $? != 0 ]; then
return $OCF_SUCCESS
fi
//调用fence_compute对计算节点进行Evacuate操作
fence_compute ${fence_options} -o off -n $node
rc=$?
//如果Evacuate操作成功, 则重置此节点的evacuate属性值为“no”
if [ $rc = 0 ]; then
update_evacuation ${node} no
ocf_log notice "Completed evacuation of $node"
//如果Evacuate操作失败, 则重置此节点的evacuate属性值为“yes”
else
ocf_log warn "Evacuation of $node failed: $rc"
update_evacuation ${node} yes
fi
fi
done
return $OCF_SUCCESS
}

```

在 `handle_evacuation` 函数中调用的另外一个函数就是 `update_evacuation` 函数, 该函数的功能很简单, 就是根据传入的节点主机名参数和 `evacuate` 属性值参数, 通过 `attrd-updater` 命令行工具重置节点的 `evacuate` 属性值, `update_evacuation` 函数的实现过程如下:

```

update_evacuation() {
//设置节点evacuate属性值, ${1}为节点名, ${2}为evacuate属性值
attrd_updater -p -n evacute -Q -N ${1} -v ${2}
arc=$?
if [ ${arc} != 0 ]; then
    ocf_log warn "Can not set evacuation state of ${1} to ${2}: ${arc}"
fi
return ${arc}
}

```

通过 Fence_compute 和 NovaEvacuate 的分析, 可以看出正是这两个资源代理之间的相互协调工作, 才实现了对 OpenStack 计算节点的监控和实例迁移功能。其中, fence_compute 负责监控 Pacemaker 集群中是否存在故障节点, 并通过节点 evacuate 属性为故障节点打上标记。NovaEvacuate 则负责实施跟踪监控集群中是否有节点被标记为故障状态 (evacuate 属性值被设为 “yes”), 并对已标记为故障状态的计算节点进行实例迁移和迁移后的节点属性更新操作。

13.3.4 计算节点高可用实现原理与问题诊断分析

在初期基于 Pacemaker 的 OpenStack 高可用集群部署中, 为了监控计算节点以及运行在其上 OpenStack 相关服务的健康状况, 同时为了克服 Corosync 节点水平扩展的限制, 必须为每个计算节点创建单节点集群。然而, 现在的 Pacemaker 集群引入 Pacemaker_remote 组件后, 已经克服了 Corosync 的节点限制, 并允许 OpenStack 控制节点和计算节点同时加入一个完整的 Pacemaker 集群中。整个 Pacemaker 集群采用统一的控制域, 计算节点上的服务由 Pacemaker 集群控制层面进行管理和驱动, 且与控制节点不同, 计算节点不再是完整的 Pacemaker 集群成员, 因此也不用运行全栈集群软件, 而仅需运行 Pacemaker_remote 软件即可, 通过统一控制域的操作, Pacemaker 对运行在故障计算节点或即将运行在故障计算节点上的实例发起自动恢复操作。基于 Pacemaker/Pacemaker_remote 的 OpenStack 计算节点高可用技术实现过程如下:

- ❑ Pacemaker 实时监控其与计算节点 Pacemaker_remoted 进程的连通性, 从而确认计算节点是否正常运行, 如果不能与 Pacemaker_remoted 建立会话连接, 则触发集群恢复机制。
- ❑ Pacemaker 通过 Pacemaker_remoted 依次启动计算节点上的服务, 启动顺序通过 Pacemaker 的 Order 约束由用户自定义。
- ❑ 如果计算节点上的某个服务启动失败, 则依赖该服务的其他全部服务均不会启动, 直至该服务维护后成功启动为止。
- ❑ 如果某个服务不能正常停止, 则运行该服务的计算节点将被 Fencing, 从而保证集群数据的完整性。
- ❑ 如果计算节点服务的健康检查失败, 则该资源以及依赖该资源的全部其他资源都

被停止并重新启动。在这个过程中，如果服务停止失败，则同样会触发集群的 Fencing 机制从而隔离该计算节点。

- ❑ 计算节点集群资源的成功重启不会对实例本身的运行产生影响，仅会造成实例网络连接的短暂中断。

基于上述实现过程，可以利用 Pacemaker 的节点监控和隔离功能，并最终调用 Nova 的 host-evacuate API 进行故障计算节点上实例的撤离，并将其在其他正常计算节点上进行恢复。当某个计算节点出现故障时，Pacemaker 将进行以下操作：

1) 执行 nova service-disable 操作，停用计算节点 Nova 服务从而在对其进行 Fencing 之前将其从 Nova 的 Scheduler 调度后端中移除。对于实例 HA 而言，这不是必须的操作，但是此步骤可以加速计算节点故障期间的虚拟机创建调度过程，因为在虚拟机创建调度过程中，Nova 不用再主动判断计算节点已经故障并重新将虚拟机创建请求重新调度到其他计算节点。

2) 对故障计算节点进行 Fencing 操作，即通过网络直接 Power off 计算节点。此步骤为必须的关键操作，Pacemaker 需要通过节点 Fencing 操作确保问题节点完全被隔离。

3) 执行 fence_compute off，等待 Nova 确认故障计算节点已经处于 down 状态，同时调用 Nova 的 host-evacuate API 进行虚拟机迁移。

4) 执行 fence_compute on，默认不进行其他操作，除非计算节点被 Fencing 后又重新启动恢复。

5) 当计算节点重新恢复后，执行 nova service-enable 操作，告知 Nova 的 Scheduler 计算节点已经恢复，可以对其进行调度。同步骤 1 类似，此步骤也并非必须执行的操作。

在上述步骤中，需要用户进行具体实现的主要是步骤 3，即 Nova 计算节点服务已完全处于 down 状态的监控。在 Redhat 发行的 OpenStack RDO 版本中，Nova 计算服务的监控主要由 fence_compute 代理程序来实现，在 Pacemaker 集群中，该步骤由基于 fence_compute 隔离代理程序的 Stonith 来实现，通常将其命名为 fence-nova，其创建过程如下：

```
pcs stonith create fence-nova fence_computeauth-url=\
http://$vip_keystone:35357/v2.0/ login=admin passwd=admin\
tenant-name=admin record-only=1 --force
```

在生产环境中，步骤 2 通常由物理计算节点的 Fencing 设备来完成，以最为常见的 IPMI 为例，要实现 Pacemaker 对计算节点的物理 Fencing 操作，需要在 Pacemaker 中针对每个计算节点创建一个对应的 Stonith 设备，该设备的 Fencing 功能由 fence_ipmilan 代理程序实现。假如需要对 IPMI 管理 IP 为 192.168.142.100 的计算节点 Compute1 在 Pacemaker 集群中创建 Fencing 设备，则其创建过程如下：

```
pcs stonith create ipmi-compute1 fence_ipmilan pcmk_host_list=compute1\
ipaddr=192.168.142.100 login=admin passwd=admin lanplus=1 \
cipher=1 op monitor interval=60s
```


如果有多个计算节点，则对应创建多个物理 Fencing 设备。对每个计算节点，将其对应的物理 Fencing 设备与 fence-nova 添加到 Level 为 1 的 Stonith 组中。如针对 Compute1 计算节点，将其对应的物理 Fencing 设备 ipmilan-compute1 与 fence-nova 添加到 Level 为 1 的 Stonith 中，添加过程如下：

```
pcs stonith level add 1 compute1 ipmilan-compute1,fence-nova
```

在实际运行中，Pacemaker 会周期性地执行 fence-nova，以实时监控 OpenStack 集群中的计算节点 Nova 服务运行状态，一旦监控到某个计算节点的 Nova-compute 服务处于 down 状态，则在 Pacemaker 集群中为该计算节点设置值为“yes”的 evacuate 属性。此外，Pacemaker 集群中的另一个 OCF 类型资源代理 NovaEvacuate 也在周期性地监测集群中是否有节点被设置了 evacuate 属性，并检查针对该节点的 evacuate 属性值是否为“yes”。如果发现集群中有节点 evacuate 属性为“yes”，并且该节点属于计算节点，则以命令行方式调用 fence_compute 对该计算节点上的实例进行 evacuate 操作。在实际部署中，基于 NovaEvacuate 资源代理的 Pacemaker 资源通常命名为 nova-evacuate，其创建过程如下：

```
pcs resource create nova-evacuate ocf:openstack:NovaEvacuate\
auth_url=http://$vip_keystone:35357 username=admin password=admin \
tenant_name=admin --force
```

在 Pacemaker 集群中，Stonith 设备 fence-nova 和 Pacemaker 资源 nova-evacuate 均运行在控制层面上，fence-nova 和 nova-evacuate 可以在任一控制节点上运行，如果运行 fence-nova 或 nova-evacuate 的计算节点故障，则 Pacemaker 将在其他控制节点上重新启动 fence-nova 或 nova-evacuate。基于 Pacemaker 的 OpenStack 高可用集群计算节点高可用实现原理如图 13-21 所示，其中 Pacemaker 集群逻辑上被划分为控制层面和计算层面，控制层面由运行 Pacemaker 和 Corosync 的三个控制节点组成，同时 fence-nova 和 nova-evacuate 运行在控制层面中，针对每个计算节点，其对应的物理 Fencing 设备与 fence-nova 属于 Level 为 1 的 Stonith 组。计算层面由运行 Pacemaker_remote 的 OpenStack 计算节点组成，除了集群软件 Pacemaker_remote，计算节点还运行 Nova-compute、OpenVswitch agent、Ceilometer-compute 以及虚拟化支持的 Libvirt 服务，而这些服务全部通过 Pacemaker/Pacemaker_remote 进行管理控制。

现假设计算节点 Compute1 故障，则位于该计算节点上的虚拟机迁移过程将按照图 13-21 中编号顺序依次进行，虚拟机迁移各个步骤解释如下：

1) 计算节点 Compute1 故障并被 Pacemaker 隔离；

2) fence-nova 监控到计算节点 Compute1 上的 Nova 服务处于 down 状态，通过集群属性设置命令 attrd_updater 将 Compute1 在 Pacemaker 集群中的 evacuate 属性设置为“yes”，以便 nova-evacuate 可以检测到该属性值。节点属性设置命令如下：

```
attrd_updater -p -n evacuate -Q -N compute1 -v yes
```

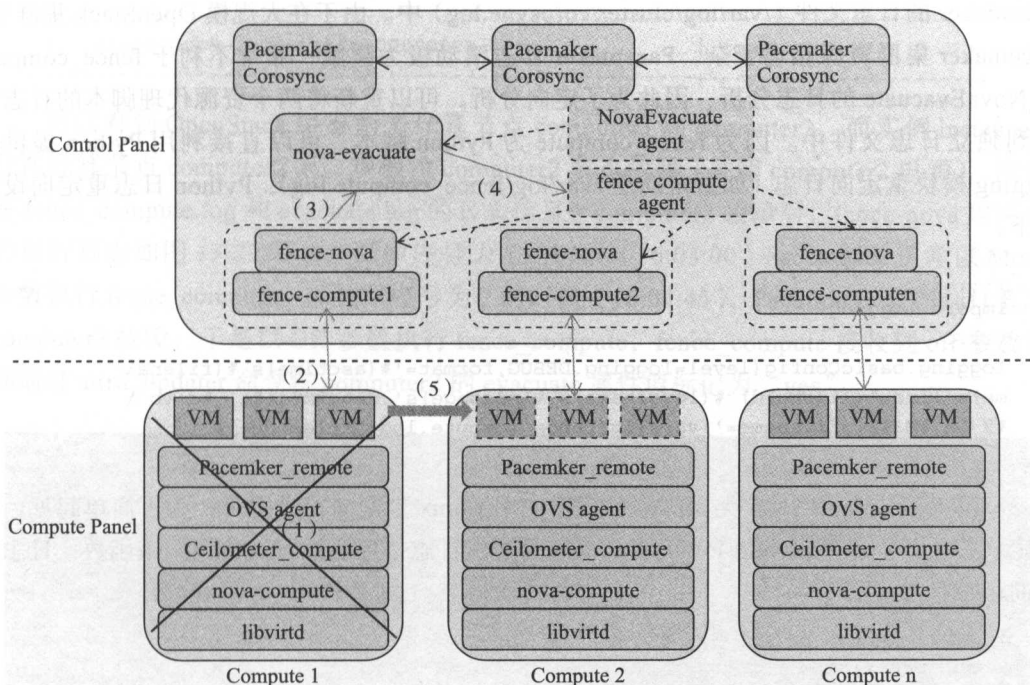


图 13-21 基于 Pacemaker 的 OpenStack 高可用集群计算节点高可用原理图

3) nova-evacuate 监测到 Pacemaker 集群中 Compute1 计算节点的 evacuate 属性为“yes”，nova-evacuate 判定 Compute1 计算节点需要进行 Evacuate 操作。nova-evacuate 监测集群中所有节点 evacuate 属性的命令如下：

```
attrd_updater -n evacuate -A
```

4) nova-evacuate 以命令行方式调用 fence_compute 对 Compute1 节点进行 Evacuate 操作，fence_compute 命令行调用方式如下：

```
fence_compute -o off auth_url=http://$vip_keystone:35357 \
username=admin password=admin tenant_name=admin -n compute1
```

5) fence_compute 调用 Nova 的 evacuate API 对 Compute1 计算节点上的实例进行 Evacuate 迁移操作，操作成功后，nova-evacuate 将重置 Compute1 的 evacuate 属性值为“no”。

在基于 Pacemaker/Pacemaker_remote 实现的 OpenStack 高可用集群实例 HA 方案中，Fence_compute 和 NovaEvacuate 资源代理脚本起着故障监控和实例恢复的关键作用。为了进一步分析计算节点实例高可用实现的具体原理和过程，下面通过跟踪 fence_compute 和 NovaEvacuate 资源代理脚本的执行日志来分析计算节点故障时实例自动撤离迁移的过程。正常情况下，fence_compute 和 NovaEvacuate 以 Pacemaker 隔离设备 fence-nova 和资源 nova-evacuate 的形式运行在 Pacemaker 集群中，因此相关的运行日志被定向到

Pacemaker 的日志文件 (/var/log/cluster/corosync.log) 中。由于在大规模 OpenStack 集群中, Pacemaker 集群资源相当繁杂, Pacemaker 日志滚动极为频繁, 非常不利于 fence_compute 和 NovaEvacuate 的日志分析, 因此为了定向分析, 可以重新将两个资源代理脚本的日志定向到独立日志文件中。因为 fence_compute 为 Python 脚本, 可以直接利用 Python 提供的 logging 模块重定向日志 (如重定向至 /var/log/fence_compute.log), Python 日志重定向设置如下:

```
.....
import logging
.....
logging.basicConfig(level=logging.DEBUG,format='%(asctime)s %(filename)\n
me)s[line:%(lineno)d] %(levelname)s %(message)s',datefmt='%a, %d %b \
%Y %H:%M:%S',filename='/var/log/fence_compute.log',filemode='w')
.....
```

NovaEvacuate 为 OCF 类型脚本, 直接使用 Linux 重定向符号即可, 为了简单起见, 可以定义一个日志重定向函数 log_info, 在需要输出日志信息的地方直接调用该函数, 日志重定向函数可参考如下:

```
function log_info ()
{
    DATE_N=`date "+%Y-%m-%d %H:%M:%S"`
    USER_N=`whoami`
    echo "${DATE_N} ${USER_N} [INFO] ${1}" >>/var/log/evacuate.log
}
```

日志重定向设置完成后, 在 Python 脚本 fence_compute 中, 调用 logging.debug("message") 即可将 “messages” 输出到 /var/log/fence_compute.log 中, 而在 NovaEvacuate 中, 调用 log_info ("messages") 即可将 “messages” 输出到 /var/log/evacuate.log 中。现在, 通过跟踪这两个日志文件即可看到 OpenStack 实例 HA 的过程, 在当前 OpenStack 集群中, 实例和计算节点主机情况信息如下:

```
[root@controller1-vm ~]$ nova hypervisor-list
+-----+-----+-----+-----+
| ID | Hypervisor hostname | State | Status |
+-----+-----+-----+-----+
| 1 | computer1 | up | enabled |
| 4 | computer2 | up | enabled |
+-----+-----+-----+-----+

[root@controller1-vm ~]$ nova hypervisor-servers computer1
+-----+-----+-----+-----+
| ID | Name | Hypervisor ID | Hypervisor Hostname |
+-----+-----+-----+-----+
+-----+-----+-----+-----+

[root@controller1-vm ~]$ nova hypervisor-servers computer2
+-----+-----+-----+-----+
| ID | Name | Hypervisor ID | Hypervisor Hostname |
```

```

+-----+-----+-----+-----+
|...d1e3e72cb9 | instance-00000001 | 4 | computer2 |
+-----+-----+-----+-----+

```

可以看到 OpenStack 中有两个计算节点 computer1 和 computer2，而实例 instance1 运行在计算节点 computer2 上。现假设 computer2 节点故障（关闭 computer2 电源），注意观察 fence_compute.log 和 evacuate.log 的日志信息。computer2 故障后，fence-nova 隔离设备的运行日志如图 13-22 所示，图中序号为 1 的时候（16:03:00）Pacemaker 正常以 Monitor 参数执行 fence_compute，而到了序号为 2 的时候（16:03:45），Pacemaker 监测到计算节点 computer2 故障，于是以 Off 参数执行 fence_compute，fence_compute 接收到 off 参数后立即通过 attrd_updater 命令将 computer2 的 evacuate 属性值标记为“yes”。

```

[root@controller1-vm ~]# tail -f /var/log/fence_compute.log
Mon, 02 Jan 2017 16:00:57 fence_compute[line:203] DEBUG Pacemaker stonith start executing fence_compute!
Mon, 02 Jan 2017 16:00:58 fence_compute[line:246] DEBUG Because of the vaule of --action is STATUS/MONITOR,There is nothing to do
tail: /var/log/fence_compute.log: file truncated
Mon, 02 Jan 2017 16:01:58 fence_compute[line:246] DEBUG Because of the vaule of --action is STATUS/MONITOR,There is nothing to do
tail: /var/log/fence_compute.log: file truncated
Mon, 02 Jan 2017 16:03:00 fence_compute[line:246] DEBUG Because of the vaule of --action is STATUS/MONITOR,There is nothing to do 1
tail: /var/log/fence_compute.log: file truncated
Mon, 02 Jan 2017 16:03:36 fence_compute[line:132] DEBUG Get hypervisors list,if there is domain option
Mon, 02 Jan 2017 16:03:36 connectionpool.py[line:203] INFO Starting new HTTP connection (1): 192.168.142.203
Mon, 02 Jan 2017 16:03:37 connectionpool.py[line:383] DEBUG "POST /v2.0/tokens HTTP/1.1" 200 3667
Mon, 02 Jan 2017 16:03:37 connectionpool.py[line:203] INFO Starting new HTTP connection (1): 192.168.142.210
Mon, 02 Jan 2017 16:03:37 connectionpool.py[line:383] DEBUG "GET /v2/5751ec6ea581413484cad0bed9c39bf4/os-hypervisors/detail HTTP/1.1" 200 2651
tail: /var/log/fence_compute.log: file truncated
Mon, 02 Jan 2017 16:03:45 fence_compute[line:241] DEBUG Because of the vaule of --action is OFF/REBOOT,Setting evacuation is YES! 2
Mon, 02 Jan 2017 16:03:45 fence_compute[line:87] DEBUG Setting fencing status for computer2 to yes
Mon, 02 Jan 2017 16:03:45 fencing.py[line:1199] INFO Executing: attrd_updater -p -n evacuate -Q -N computer2 -v yes
Mon, 02 Jan 2017 16:03:45 fencing.py[line:1219] DEBUG 0 3

```

图 13-22 计算节点 computer2 故障后 fence_compute 日志

nova-evacuate 资源的运行日志如图 13-23 所示，图中序号为 1 的时候（16:03:38），Nova Evacuate 没有监测到集群中任何节点被标记为需要 Evacuate，而在序号为 2 的时候（16:03:48），NovaEvacuate 监测到 computer2 被标记为需要 Evacuate，序号为 3 的地方开始对 computer2 节点执行 Evacuate 操作，序号为 4 的时候（16:04:12），针对 computer2 计算节点的实例 Evacuate 操作成功完成，并将 computer2 的 evacuate 属性值更新为“no”，序号为 5 和 6 的时候 NovaEvacuate 确认 computer2 不再需要执行 Evacuate，即计算节点 computer2 上的实例已经全部迁移撤离完成。

实例自动迁移完成后，通过 Nova 命令行可以确认当前实例位于哪个 Hypervisor，即哪个计算节点上。因为之前的实例 instance1 位于故障计算节点 computer2 上，因此迁移后实例应该位于 computer1 计算节点上，验证结果如下所示：

```

[root@controller1-vm ~]$ nova hypervisor-servers computer2
+-----+-----+-----+-----+
| ID | Name | Hypervisor ID | Hypervisor Hostname |
+-----+-----+-----+-----+
[root@controller1-vm ~]$ nova hypervisor-servers computer1

```

ID	Name	Hypervisor ID	Hypervisor Hostname
...dle3e72cb9	instance-00000001	4	computer1

```

2017-01-02 16:03:38 root [INFO] 1.begining evacuate_validate!
2017-01-02 16:03:38 root [INFO] the fence_options is: -k http://192.168.142.203:35357/v2.0/ -l admin -p admin -t admin
2017-01-02 16:03:38 root [INFO] 2.begining evacuate_monitor!Pacemaker execute NovaEvacuate with Monitor option
2017-01-02 16:03:38 root [INFO] Current Pacemaker cluster has evacuation attribute as follow(attrd_updater -n evacuate -A):
could not query value of evacuate: attribute does not exist (1)
2017-01-02 16:03:38 root [INFO] Call handle_evacuations function to handle evacuation!
2017-01-02 16:03:38 root [INFO] 3.Begining handle evacuation:node is ,current evacuation is ,and will entry into while loop!
2017-01-02 16:03:38 root [INFO] Now jump out the while loop!
2017-01-02 16:03:48 root [INFO] 1.begining evacuate_validate!
2017-01-02 16:03:48 root [INFO] the fence_options is: -k http://192.168.142.203:35357/v2.0/ -l admin -p admin -t admin
2017-01-02 16:03:48 root [INFO] 2.begining evacuate_monitor!Pacemaker execute NovaEvacuate with Monitor option
2017-01-02 16:03:48 root [INFO] Current Pacemaker cluster has evacuation attribute as follow(attrd_updater -n evacuate -A):
name="evacuate" host="computer2" value="yes" (2)
2017-01-02 16:03:48 root [INFO] Call handle_evacuations function to handle evacuation!
2017-01-02 16:03:48 root [INFO] 3.Begining handle evacuation:node is computer2,current evacuation is yes,and will entry into while loop!
2017-01-02 16:03:48 root [INFO] The state of computer2 evacuation is yes,computer2 need evacuate!
2017-01-02 16:03:48 root [INFO] Initiating evacuation of computer2 (3)
2017-01-02 16:03:50 root [INFO] Update evacuation of computer2 from YES to controller2-vm@1483344230
2017-01-02 16:03:50 root [INFO] 4.update Node computer2 Evacuation to be controller2-vm@1483344230
2017-01-02 16:03:50 root [INFO] Mon Jan 2 16:03:50 CST 2017starting run fence_compute --k http://192.168.142.203:35357/v2.0/ -l admin -
uter2
2017-01-02 16:04:12 root [INFO] 4.update Node computer2 Evacuation to be no (4)
2017-01-02 16:04:12 root [INFO] Completed evacuation of computer2
2017-01-02 16:04:12 root [INFO] Now jump out the while loop!
2017-01-02 16:04:23 root [INFO] 1.begining evacuate_validate!
2017-01-02 16:04:23 root [INFO] the fence_options is: -k http://192.168.142.203:35357/v2.0/ -l admin -p admin -t admin
2017-01-02 16:04:23 root [INFO] 2.begining evacuate_monitor!Pacemaker execute NovaEvacuate with Monitor option
2017-01-02 16:04:23 root [INFO] Current Pacemaker cluster has evacuation attribute as follow(attrd_updater -n evacuate -A):
name="evacuate" host="computer2" value="no" (5)
2017-01-02 16:04:23 root [INFO] Call handle_evacuations function to handle evacuation!
2017-01-02 16:04:23 root [INFO] 3.Begining handle evacuation:node is computer2,current evacuation is no,and will entry into while loop!
2017-01-02 16:04:23 root [INFO] computer2 is either fine or already handled (6)
2017-01-02 16:04:23 root [INFO] Now jump out the while loop!

```

图 13-23 计算节点 computer2 故障后 NovaEvacuate 日志

可以看到，在 computer2 计算节点故障后，通过 OpenStack 高可用集群中的实例 HA 方案，实例 instance1 (ID 以 2cb9 结尾) 被自动迁移至 computer1 计算节点启动运行。当 computer2 重新启动并加入 Pacemaker 高可用集群后，其可以正常接受 Nova-scheduler 新的实例创建请求调度，但是原有实例均不会在其上重新启动。在实际部署使用中，当计算节点故障时，如果其上的实例没有进行自动迁移，则可按照如下步骤进行诊断：

(1) 检查 Pacemaker 集群资源是否正常运行

与 Nova 实例相关的集群资源服务主要有位于控制节点的 nova-api、nova-scheduler、nova-conductor 服务，以及位于计算节点的 nova-compute 和 pacemaker_remote 服务，要实现实例的 HA，集群中的 nova-scheduler 服务需要发现可用的计算节点，因此必须保证上述服务至少在某个控制节点或计算节点上处于 Active 状态。

(2) 检查 nova-evacuate 资源运行状态

nova-evacuate 资源由 NovaEvacuate 资源代理控制，NovaEvacuate 的作用在于实时跟踪集群全部节点的 evacuate 属性，一旦发现有节点 evacuate 属性被设置为“yes”，则马上启动实例撤离操作。因此，实例 HA 功能要能够正常实现，nova-evacuate 资源必须正常运行

在某个控制节点上。

(3) 检查 fence-nova Fence 设备运行状况

fence-nova 由 fence_compute 资源代理脚本控制，fence-nova 通常与计算节点物理 fence 设备处于同一个隔离组中，fence-nova 资源实时监控计算节点 Nova 服务是否已经完全故障，并为 Nova 服务已故障的计算节点设置 evacuate 属性，换句话说，如果 fence-nova 不能监控到计算节点 Nova 服务的故障，则实例 HA 功能不可能实现。因此，必须确保 fence-nova 在某个控制节点上正常运行。

(4) 检查故障计算节点是否已被完全隔离

在 Pacemaker 集群中，如果计算节点资源被发现存在异常行为，则 Pacemaker 将调用 fence 设备对该节点进行隔离，节点被隔离后，其上的全部服务都被强制停止。如果故障节点隔离未成功，请检查集群或节点 fence 配置。

(5) 调试资源代理源代码

如果全部 Pacemaker 资源和 Stonith 隔离设备均正常运行，但是仍然不能实现实例高可用，则可能需要跟踪调试与实例 HA 相关的资源代理脚本程序（调试方法可以参考本节前文内容），两个核心的资源代理脚本分别是基于 shell 的 NovaEvacuate 和基于 Python 的 fence_compute，例如笔者在初次部署时安装的资源代理软件 RPM 包的名称为 resource-agents-3.9.5-54.el7_2.9.x86_64.rpm，在该软件包提供的 fence_compute 脚本中，为故障计算节点设置 evacuate 属性值的函数语句如下：

```
def set_attrd_status(host, status, options):
    logging.debug("setting fencing status for %s to %s" % (host, status))
    run_command(options, "attrd_updater -p -n evacuate -Q -N %s -v %s" % (host, status))
```

上述脚本中，attrd_updater 命令行参数中的“evacute”漏写了字母“a”，正确写法应该为“evacuate”。由于此处为节点设置的属性名称为 evacuate，而 NovaEvacuate 中实时查询的属性却是 evacuate，因此在计算节点故障时，尽管 fence_compute 已经监控到并且设置了对应的故障标志属性，但是由于 fence_compute 与 NovaEvacuate 使用了不同的属性名称（fence_compute 使用的是“evacute”，而 NovaEvacuate 使用的是“evacuate”），最终 NovaEvacuate 仍然认为集群中没有计算节点需要进行实例迁移操作，因此故障计算节点上的实例自然不会被迁移。当更新了 resource-agents 软件包或将 fence_compute 中的“evacute”更正为“evacuate”后，实例 HA 功能正常实现。

13.4 OpenStack Neutron 网络理解与故障问题诊断

13.4.1 OpenStack Neutron 网络概念基础

OpenStack 中的网络服务可以由 Nova-network 或 Neutron 项目提供，随着 OpenStack

新版本的不断更替，传统 Nova-network 服务的功能更新和用户越来越少，而 Neutron 服务不断发展和壮大，目前已成为 OpenStack 的网络核心项目。Neutron 项目的目标是实现云计算环境中的“网络即服务 (NaaS)”，因而 Neutron 网络的实现过程充分遵循了软件定义网络 SDN 的原则，利用 Linux 系统中的各种网络虚拟化技术，Neutron 实现了 OpenStack 云计算环境中用户自定义网络的功能。为了实现网络即服务的功能，Neutron 整合利用了 Linux 系统中各种网络虚拟化技术，因此在分析 Neutron 网络之前，了解 Linux 系统中各种网络技术和术语是十分必要的，下面对 Neutron 网络中常见的网络技术和术语进行简单的描述。

- ❑ Linux 网桥 (Bridge)：Bridge 是虚拟机网络通信的基础，在 Linux 系统中，Bridge 类似物理交换机，是可以连接不同网络设备的虚拟设备，可以简单地将其看作一个虚拟 Hub 或交换机设备。在 Linux 系统中，使用 `brctl show` 命令可以查看系统中的 Bridge。
- ❑ 集成网桥 (br-int)：br-int 是 Bridge-integration 的简写，在 Neutron 网络中，br-int 通常用以连接 Linux 网桥和隧道网桥，并进行不同网络 ID 之间的映射转换。在不同的网络模式（如 GRE 或 VLAN/VxLAN 模式）中，br-int 实现的功能有所不同。
- ❑ 外部网桥 (br-ex)：br-ex 是 Bridge-external 的简写，在 Neutron 网络中，br-ex 通常表示与节点外部网络连接的网桥，即所有租户实例网络都需要经由 br-ex 才能与外部通信。
- ❑ 隧道网桥 (br-tun)：br-tun 是 Bridge-tunnel 的简写，只有在 GRE 网络模式中才会存在 br-tun（如果是 VLAN 网络模式则是 br-eth 网桥），br-tun 主要负责网络 ID 与隧道 ID 之间的映射转换。
- ❑ GRE：通用路由封装 (Generic Routing Encapsulation, GRE) 是一种路由协议封装方式，通过 GRE 路由封装隧道，节点之间可以进行不同的网络通信。在 Neutron 网络中，通常是基于 L3 的 GRE 网络模式，并结合 br-tun 网桥来实现。
- ❑ VxLAN：使用 UDP 作为底层传输协议的一种 Overlay 网络技术实现，由于 VLAN 网络的 ID 范围在 1~4094 之间，在大型网络环境中 ID 资源有被耗尽的风险，因此 VxLAN 通常被认为是随 VLAN 的扩展或替换。
- ❑ TAP：TAP (Test Access Device) 设备是一个虚拟二层网络接口设备，类似 Linux 系统中的物理网口，可以接收和发送网络数据。
- ❑ VETH：VETH 是虚拟 Ethernet 设备，VETH 设备总是成对出现，向其一端输入数据，VETH 会改变数据的方向并将其送入内核网络核心，完成数据的注入，之后在另一端便能读到此数据，简单地说，从 VETH 设备一端输入的数据总是会从另一端输出。在 Neutron 中，两个不同网桥之间通常使用 VETH 对进行数据传输。
- ❑ qbrxxx：Linux 网桥 (Bridge) 设备，qbrxxx 位于实例和 br-int 网桥之间，主要负责网络安全组 (Security Group) 规则设置。

- ❑ qvovxxx : Neutron 的 VETH 设备, qvo 表示 OpenVswitch 一侧的 veth 设备, qvo 各个字母的解释为: q-quantum, v-veth, o-openvswitch (quantum 是 Neutron 的前身)。
- ❑ qvbvxxx : Neutron 的 VETH 设备, qvb 表示 Linux Bridge 一侧的 veth 设备, qvb 各个字母解释为: q-quantum, v-veth, b-bridge。
- ❑ qrvxxx : L3 路由命名空间内部租户子网网关接口, 同时位于集成网桥 br-int 上 (使用 neutron router-interface-add 命令为 L3 路由添加子网接口时出现)。不同租户网络对应着不同的 qrvxxx 接口, 租户子网网关 IP 地址通常配置在 qrvxxx 接口上。
- ❑ qgxxx : L3 路由命名空间内部外网网关接口, 同时位于外部网桥 br-ex 上 (使用 neutron router-gateway-set 命令为 L3 路由设置外网网关时出现), 每个 L3 虚拟路由有且仅有一个 qgxxx 接口, 租户网络通过 qrvxxx 接口进入 L3 路由, 经过 L3 路由后由 qgxxx 接口进入外部网络。
- ❑ NameSpace : 命名空间, Linux 系统中资源隔离的一种实现机制, 位于不同命名空间中的网络对象彼此不可见, 因此不同命名空间中可以使用相同的变量或对象, 如相同的 IP 地址。在 Neutron 网络中, L3 agent 和 DHCP agent 均使用命名空间实现隔离。

相对于传统的 Nova-network, 由于 Neutron 网络插件式的架构设计, 使其具有更强的灵活性。使用 Neutron 网络, 用户不仅可以使⽤各个网络设备厂商提供的网络解决方案, 还可使⽤ Linux 系统原生的网络虚拟化技术, 当⽤户使⽤不同的网络插件时, 对应的 Neutron 网络流图也不相同。在 OpenStack 社区中, 用户部署使⽤最多的是 Open vSwitch (OVS) 插件, 网络模式使⽤较多的是 VLAN 和 GRE/VxLAN 模式, 因此本节将重点介绍基于 OVS 插件的 VLAN 和 GRE 模式 Neutron 网络 (VxLAN 模式与 GRE 模式网络架构类似, 不同之处在于隧道封装协议)。图 13-24 为 GRE 模式下 Neutron 网络的数据流图, 当实例 VM1 访问外部网络时, VM1 的数据进⼊计算节点 TAP 设备 (A), 经过 veth 对 A-B 后进⼊ Linux 网桥 qbr-xxx, 数据在 qbr 中进⼊安全规则处理后, 通过另⼀对 veth 设备 C-D (C 通常命名为 qvb-xxx, D 通常命名为 qvo-xxx) 进⼊ OVS 提供的集成网桥 br-int, 数据在 br-int 中经过外部 Vlan Tag 与内部 Vlan Tag 的转换后, 通过 veth 对 E-F (E 命名为 patch-tun, F 命名为 patch-int) 进⼊隧道网桥 br-tun, 数据在 br-tun 中经过 Vlan Tag 与 Tunnel ID 的转换后, 通过物理节点之间的网络连接 G-H 进⼊网络节点隧道网桥 br-tun, 网络节点 br-tun 对数据进⼊ Tunnel ID 与 Vlan Tag 的转换后, 通过 veth 对 I-J 进⼊网络节点 br-int 网桥, 数据在 br-int 网桥中经过内部 Vlan Tag 与外部 Vlan Tag 转换后, 通过 TAP 设备进⼊ L3 路由命名空间中的 qrvxxx 接口 (M-N), 数据在 L3 路由中经过路由表处理后, 由路由 qgxxx 接口进⼊ br-ex 网桥 (K-L), 最终数据进⼊外部网络。当外网对 VM1 进⼊访问时, Neutron 数据流过程与此相反。

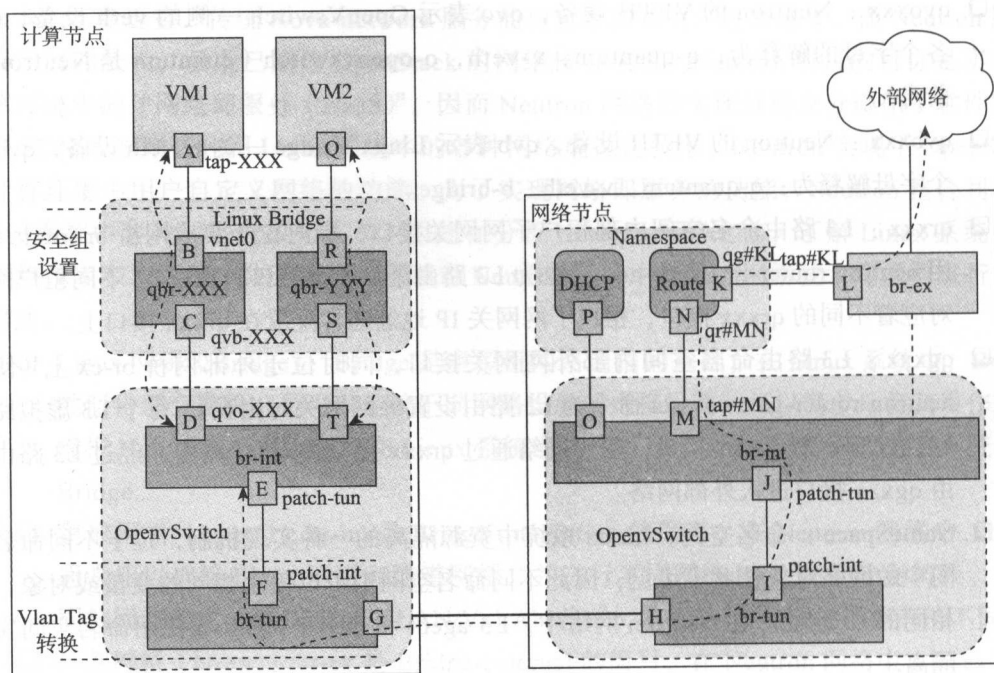


图 13-24 GRE 模式下 Neutron 网络简化流程图

图 13-25 为 VLAN 模式下 Neutron 网络的数据流图，VLAN 模式与 GRE 模式的不同之处在于 VLAN 通过 VLAN ID 来隔离不同网络，而节点内部 VLAN Tag 与外部 VLAN Tag 之间的转换需要 br-int 和 br-eth 网桥共同完成。即 br-int 网桥负责将内部 VLAN Tag 转换为外部 VLAN Tag，而 br-eth 网桥负责将外部 VLAN Tag 转换为内部 VLAN Tag，VLAN Tag 转换完成后，节点内部的数据流与 GRE 模式类似。

13.4.2 OpenStack Neutron 网络深入理解

OpenStack 的 Neutron 网络采用插件式的架构设计，使得其部署实施非常灵活，用户可以使用不同的插件来实现二层网络交换技术，同时 Neutron 还支持不同的网络模式，如 Flat、VLAN、GRE 和 VxLAN 等。在 Neutron 中，默认使用的 ML2 插件是 OpenvSwitch (OVS) 插件，同时 OVS 也是社区中部署使用最多的插件。而在网络拓扑模式上，普遍使用的是 VLAN 和 GRE/VxLAN 模式，因此本节将深入理解和分析基于 OVS 插件实现的 VLAN 和 GRE 模式 Neutron 网络数据流和各个网络对象的功能。在 Neutron 网络故障定位分析中，熟知 Neutron 网络的整个数据流处理过程，并清楚网络数据流所涉及的各个网络对象在其中扮演的功能角色，以及掌握跟踪数据包在各个物理 / 虚拟网络对象中的收发情况，是 OpenStack 网络故障分析的前提和基础，也仅有在深入理解 Neutron 网络内部功能结构后，才能缩小故障问题的范围，并一针见血地找出问题所在。

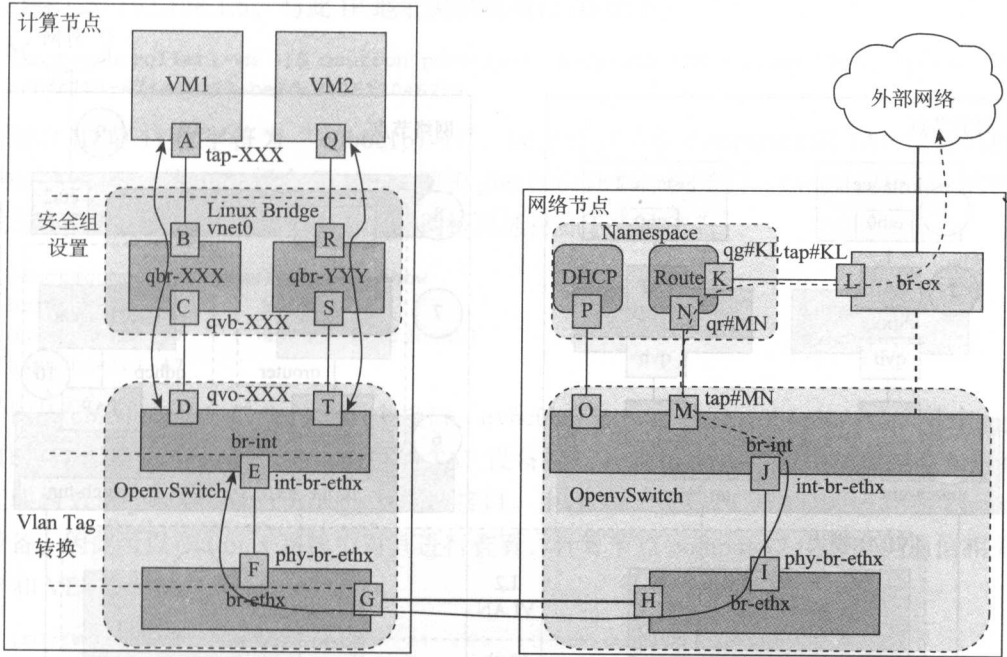


图 13-25 VLAN 模式下 Neutron 网络简化流程图

在 Neutron 网络中，实例数据通常需要经过实例虚拟网卡，计算节点 TAP 设备，Linux 网桥 qbr、qvb 和 qvo VETH 设备，br-int 网桥，patch-tun 和 patch-int VETH 设备（或 int-br-eth 和 phy-br-eth VETH 设备）和 br-tun（或 br-eth 网桥）才能进入网络节点。进入网络节点后，又经过 br-tun（或 br-eth 网桥）、patch-tun 和 patch-int VETH 设备（或 int-br-eth 和 phy-br-eth VETH 设备）、br-int 网桥、L3 agent 和 br-ex 网桥便进入外网。本节将以图 13-26 为例，依次讲解 Neutron 网络数据流如何在计算节点和网络节点之间进行传输，由于 VLAN 和 GRE 网络模式仅在数据 Tag 处理方式上存在不同，而在其他数据处理环节上类似，因此为了便于讲解，图 13-26 将两种网络模式整合在一起，后续讲解过程中二者的不同之处将会分别指出。

1. 实例网卡 eth0

运行中的实例 instance1 需要与外网通信，而实例 instance1 的虚拟网卡 eth0 配置了租户网络 IP 地址。因此，instance1 的数据包被传递给 eth0，Nova 中的实例及租户 IP 地址如下：

```
[root@controller1-vm (keystone_admin)]$ nova list
+-----+-----+-----+-----+-----+-----+
| ID      | Name    | Status | Task State | Power State | Networks |
+-----+-----+-----+-----+-----+-----+
| ...111f | instance1 | ACTIVE | -          | Running    | admin-net=192.128.1.5 |
+-----+-----+-----+-----+-----+-----+
```

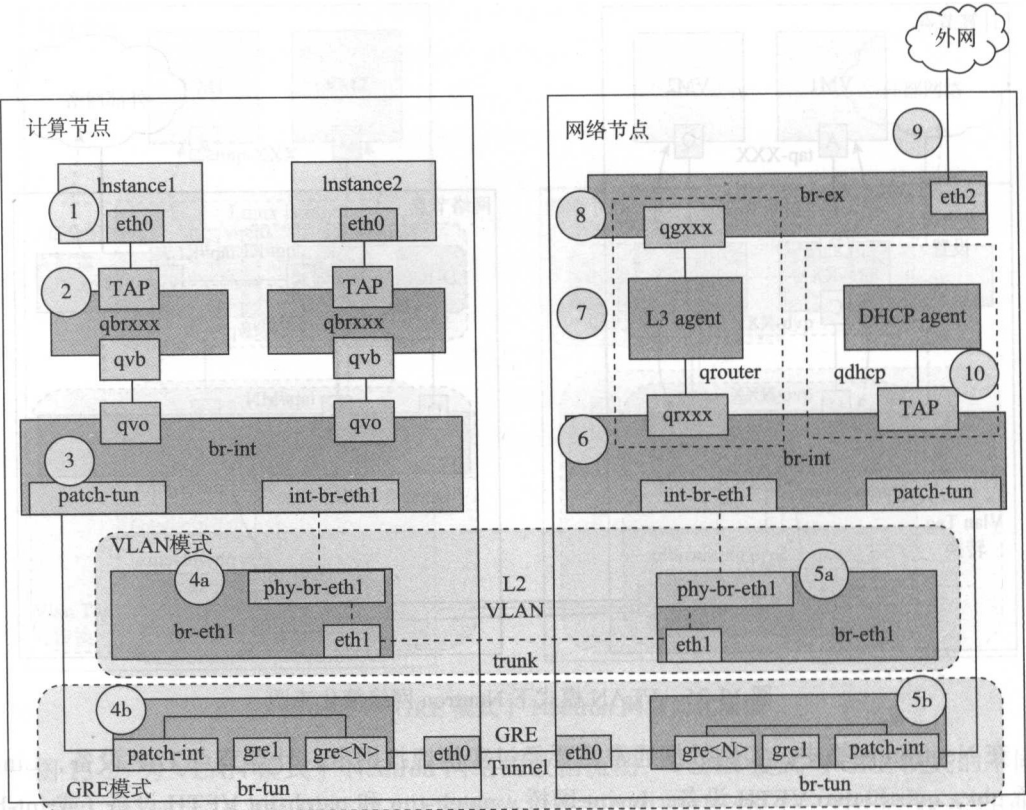


图 13-26 Neutron 网络内部数据流图

2. Linux 网桥 qbr

instance1 上虚拟网卡设备 eth0 将数据传递给计算节点上的 TAP (Test Access Point) 设备, 如 tapc0b6e1b1-47。实例当前正在使用的 TAP 设备名称可以从计算节点的 /etc/libvirt/qemu/instance-xxxxxxx.xml 文件查询到, 如下:

```
//查看hypervisor上的实例名称
[root@controller1-vm ~]$ nova hypervisor-servers computer2
+-----+-----+-----+-----+
| ID          | Name                | Hypervisor ID | Hypervisor Hostname |
+-----+-----+-----+-----+
| ...b6525d111f | instance-00000006 | 4              | computer2           |
+-----+-----+-----+-----+

[root@computer2 qemu]# cd /etc/libvirt/qemu
[root@computer2 qemu]# more instance-00000003.xml | grep tap
<target dev='tapc0b6e1b1-47' />
```

TAP 设备的命名方式为 “tap” 后跟端口 ID 前 11 位字符。端口 ID 可以从 Neutron 网络数据库中查询到, 因为实例 instance1 通过 admin-net 接入租户网络, 自动获取的租户 IP

(Fixed IP) 为 192.128.1.5，与此 IP 地址关联的端口 ID 如下：

```
[root@controller1-vm ~]$ neutron port-list |grep 192.128.1.5|awk -F \| '{print $2}'
c0b6e1b1-4764-4092-be2d-2e15120462b4
```

端口 ID 前 11 位字符为 “c0b6e1b1-47”，因此计算节点 computer2 的 TAP 设备名称为 tabc0b6e1b1-47，该 TAP 设备是 Linux 网桥 qbrc0b6e1b1-47 的接口（Linux 网桥 qbr 的命名方式与 tap 设备类似），计算节点 Linux 网桥可通过如下方式查看：

```
[root@computer2 qemu]# brctl show
bridge name          bridge id          STP enabled        interfaces
qbrc0b6e1b1-47       8000.ae8a9f2dbd87  no                 qvbc0b6e1b1-47
tapc0b6e1b1-47
```

网桥 qbrc0b6e1b1-47 中包含两个接口：qvbc0b6e1b1-47 和 tapc0b6e1b1-47，其中 tapc0b6e1b1-47 即实例 instance1 接入网桥的 TAP 设备接口，而 qvbc0b6e1b1-47 则是与 br-int 网桥互联的 veth pair 在 Linux Bridge 一侧的接口。由于 TAP 设备和 VETH 设备均为 Linux 网络设备，因此可以在 Linux 系统中对其进行查看，计算节点 computer2 中与实例通信相关的 TAP 和 VETH 设备如下所示：

```
[root@computer2 ~]# ip a
.....
7: qbrc0b6e1b1-47: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
link/ether ae:8a:9f:2d:bd:87 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::ac8a:9fff:fe2d:bd87/64 scope link
    valid_lft forever preferred_lft forever
8: qvbc0b6e1b1-47@qvbc0b6e1b1-47: <BROADCAST,MULTICAST,PROMISC,UP,LOWER_UP> mtu 1500
qdisc
pfifo_fast master ovs-system state UP qlen 1000
link/ether da:03:78:30:54:84 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::d803:78ff:fe30:5484/64 scope link
    valid_lft forever preferred_lft forever
9: qvbc0b6e1b1-47@qvbc0b6e1b1-47: <BROADCAST,MULTICAST,PROMISC,UP,LOWER_UP> mtu 1500 qdisc
pfifo_fast master qbrc0b6e1b1-47 state UP qlen 1000
link/ether ae:8a:9f:2d:bd:87 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::ac8a:9fff:fe2d:bd87/64 scope link
    valid_lft forever preferred_lft forever
10: tapc0b6e1b1-47: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master
qbrc0b6e1b1-47 state UNKNOWN qlen 500
link/ether fe:16:3e:0a:68:6e brd ff:ff:ff:ff:ff:ff
    inet6 fe80::fc16:3eff:fe0a:686e/64 scope link
    valid_lft forever preferred_lft forever
```

可以看到，与 instance1 相关的全部虚拟设备接口的后缀字符组成完全相同，同时可以看到 Linux 网桥 qbr 与集成网桥 br-int 之间通信的 veth pair。在基于 OVS 插件的 Neutron 网络中，理想情况下 TAP 设备应该直接关联到 br-int 网桥上，而不是 Linux 网桥，但是由于 OpenStack 使用 iptables 实现安全组规则，而 OVS 在 TAP 设备上进行 iptables 规则处理的兼容性不好，因此需要使用 Linux 网桥来弥补 OVS 在安全组规则处理上的不足，由于使

用 Linux 网桥的首要目的在于实现防火墙规则，因此该 Linux 网桥通常也称为“防火墙网桥”^①。在运行 TAP 设备的计算节点上，可以发现与 tapc0b6e1b1-47 设备相关的防火墙规则，如下：

```
[root@computer2 ~]# iptables -S|grep tapc0b6e1b1-47
-A neutron-openvswi-FORWARD -m physdev --physdev-out tapc0b6e1b1-47 --physdev-is-bridged -m comment --comment "Direct traffic from the VM interface to the security group chain." -j neutron-openvswi-sg-chain
-A neutron-openvswi-FORWARD -m physdev --physdev-in tapc0b6e1b1-47 --physdev-is-bridged -m comment --comment "Direct traffic from the VM interface to the security group chain." -j neutron-openvswi-sg-chain
-A neutron-openvswi-INPUT -m physdev --physdev-in tapc0b6e1b1-47 --physdev-is-bridged -m comment --comment "Direct incoming traffic from VM to the security group chain." -j neutron-openvswi-oc0b6e1b1-4
-A neutron-openvswi-sg-chain -m physdev --physdev-out tapc0b6e1b1-47 --physdev-is-bridged -m comment --comment "Jump to the VM specific chain." -j neutron-openvswi-ic0b6e1b1-4
-A neutron-openvswi-sg-chain -m physdev --physdev-in tapc0b6e1b1-47 --physdev-is-bridged -m comment --comment "Jump to the VM specific chain." -j neutron-openvswi-oc0b6e1b1-4
```

上述 iptables 中，Neutron 安全组规则在 neutron-openvswi-sg-chain 中实现，而 neutron-openvswi-ic0b6e1b1-4 控制进入实例 instance1 的数据流（inbound traffic），neutron-openvswi-oc0b6e1b1-4 控制实例 instance1 外出的数据流（outbound traffic）。

3. 集成网桥 br-int

由 TAP 设备 tapc0b6e1b1-47 进入的数据流在 Linux 网桥 qbr0b6e1b1-47 中经过防火墙规则处理后，进入 VETH 设备 qvbc0b6e1b1-47 端，并由 VETH 设备的另一端 qvoc0b6e1b1-47 进入集成网桥 br-int。在 Neutron 中，每个实例对应着一个 Linux 网桥，而全部 Linux 网桥和 Neutron 创建的其他网桥（如 br-tun 或 br-eth）均连接到 br-int 上，br-int 负责为实例外出数据流添加 VLAN Tag 或为访问实例数据流剥离 VLAN Tag。在图 13-26 中，br-int 内部除了 qvo 设备，还有 patch-tun（GRE 模式下存在）和 int-br-eth1（VLAN 模式存在）设备，int-br-eth1 是 VLAN 模式下 VETH 设备在 br-int 的端口，另一端（phy-br-eth1）在 br-eth1 网桥中，而 patch-tun 是 GRE 模式下与隧道网桥 br-tun 接口（patch-int）互联的设备。br-int 内部设备并非 Linux 网络设备，因此仅能在 OVS 环境下查看，例如在 GRE 模式环境下，计算节点 computer2 中的 br-int 内部设备如下：

```
[root@computer2 ~]# ovs-vsctl show
Bridge br-int
    fail_mode: secure
    Port "qvoc0b6e1b1-47"
tag: 1
    Interface "qvoc0b6e1b1-47"
```

① 随着 ovs 版本更新和对防火墙处理规则的增强，以后的部署中可以不需要“防火墙网桥”。

```

Port patch-tun
    Interface patch-tun
type: patch
options: {peer=patch-int}
Port br-int
    Interface br-int
type: internal

```

由于这里使用的是 GRE 模式，因此仅看到与 Linux 网桥互联的 VETH 设备 qvoc0b6e1b1-47，以及与隧道网桥互联的 patch-tun 设备。在上述计算节点 computer2 的 br-int 内部，可以看到 qvo 端口上存在一个内部 VLAN Tag 标志 “tag : 1”，这意味着该 qvo 端口是关联在内部 VLAN 1 上的 Access 类型端口，源自实例且未进行 VLAN Tag 标记的数据流在到达此端口时，数据帧将被分配 VLAN ID=1 的内部 VLAN 标志，而外部访问数据在经过外部 VLAN ID/GRE ID 与内部 VLAN ID 的转换后，对应内部 VLAN ID 的数据流在 VLAN Tag 被剥离后将由此端口发出，并经过 Linux 网桥进行安全处理后到达实例。

4. br-tun 或 br-eth 网桥

在计算节点集成网桥 br-int 内部，不管用户如何定义租户网络类型，网络都经过节点内部 VLAN ID 来区分，这种网络区分的方式允许相同计算节点上的实例彼此之间可以直接通信（通过 br-int 实现），而无须再经过其他虚拟或物理网络设备的处理。内部 VLAN ID 根据其在节点内部被创建的先后顺序依次命名（Tag1、Tag2、Tag3...），不同节点内部的 VLAN ID 可能会有差异。此外，节点内部的 VLAN ID 与用户定义的租户网络或外部网络 ID 之间没有任何关系。当节点内部实例需要与外部网络通信时，取决于用户网络设置（如 GRE 模式或 VLAN 模式），需要不同的网桥（如 br-tun 或 br-eth）进行内部 VLAN ID 与外部网络 ID（GRE Tunnel ID 或 VLAN ID）的转换，例如在实例访问外部网络时，由 br-int 添加的内部 VLAN ID 需要被转换成为外部网络 VLAN ID 或 GRE Tunnel ID，而外部网络访问节点内部实例时，需要将外部网络 VLAN ID 或 GRE Tunnel ID 转换成为内部 VLAN ID。下面针对这两种不同的网络模式分析计算节点内部不同网络类型内外部网络 ID 之间的转换。

（1）VLAN 模式

在图 13-26 中，VLAN 网络模式下，节点内部 VLAN ID 与外部 VLAN ID 之间的转换需要 br-int 网桥和 br-eth1 网桥共同完成。当具有外部 VLAN ID 的数据访问计算节点实例时，外部 VLAN ID 与内部 VLAN ID 之间的转换由 br-int 网桥实现，当物理连接网桥 br-eth1 接收到带有外部 VLAN Tag 的数据后，数据由 br-eth1 的 phy-br-eth1 进入 br-int 的 int-br-eth1 端口。当 br-int 的 int-br-eth1 端口接收到带有外部 VLAN Tag 的数据帧后，br-int 网桥将外部 VLAN ID 转换为内部 VLAN ID，之后再转发到对应内部 VLAN Tag 的 qvo 端口，并通过 VETH 设备进入 Linux 网桥 qbr；当计算节点实例访问外部网络时，内部 VLAN ID 与外部 VLAN ID 的转换由物理连接网桥 br-eth1 实现，br-int 为实例数据分配内部 VLAN

ID 之后，由 phy-br-eth1 接口进入 br-eth1 网桥，br-eth1 网桥接收到带有内部 VLAN Tag 的数据帧后，将内部 VLAN ID 转换为外部 VLAN ID，并由物理网卡 eth1 进入具有相同 VLAN ID 的物理网络。VLAN 模式下的 VLAN ID 转换过程可参考官网提供的原理图，如 13-27 所示，假设计算节点内部 VLAN ID 分别为 1 和 2，而外部 VLAN ID 分别为 101 和 102。当外部网络访问实例时，外部网络 VLAN101 和 VLAN102 由物理网卡 eth1 进入 br-eth1 网桥，之后由 VETH Pair 设备 phy-br-eth1 和 int-br-eth1 进入 br-int 网桥，在经过 br-int 网桥的 VLAN ID 转换后，外部 VLAN101 被映射成为内部 VLAN1，外部 VLAN102 被映射成为内部 VLAN2，最终 VLAN Tag 为 101 的外部数据帧被转发至 br-int 网桥中具有内部 VLAN Tag 为 1 的 qvo 端口，VLAN Tag 为 102 的外部数据帧被转发至 br-int 网桥中具有内部 VLAN Tag 为 2 的 qvo 端口。当实例访问外部网络时，没有任何 VLAN Tag 的实例数据被 br-int 的 qvo 设备打上对应的内部 VLAN Tag（VLAN ID 分别为 1 和 2）后，被转发至 br-int 的 int-br-eth1 端口，并由 phy-br-eth1 端口进入物理连接网桥 br-eth1，br-eth1 网桥再将内部 VLAN1 转换成为外部 VLAN 101，内部 VLAN2 转换成为外部 VLAN102，并由物理网卡 eth1 进入外部物理网络交换机（交换机端口配置为 trunk 模式）。

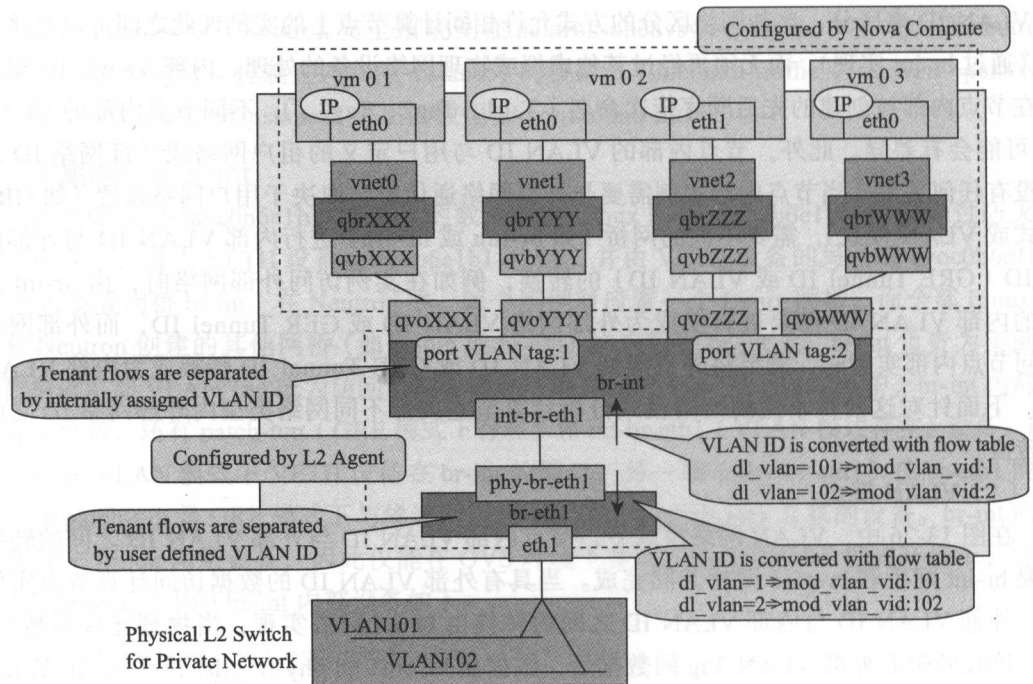


图 13-27 VLAN 模式下计算节点内外 VLAN ID 转换

(2) GRE 模式

在图 13-26 中，GRE 网络模式下，GRE 隧道 ID 与内部 VLAN ID 的转换主要由隧道网

桥 br-tun 完成。br-int 标记上内部 VLAN Tag 的实例数据被转发到 br-int 的 patch-tun 端口，并由 br-tun 上的 path-int 端口进入 br-tun 网桥。由于 br-tun 并非 Linux 网络设备，因此使用 Linux 系统命令无法查看 br-tun 网桥内部设备，而必须在 OVS 环境中才能查看。在本书介绍的 OpenStack 高可用部署模式中，计算节点 computer2 上的 br-tun 网桥内部网络设备如下：

```
[root@computer2 ~]# ovs-vsctl show
Bridge br-tun
    fail_mode: secure
    Port "gre-c0a88e70"
        Interface "gre-c0a88e70"
type: gre
options: {df_default="true", in_key=flow, local_ip="192.168.142.35",
out_key=flow, remote_ip="192.168.142.112"}
    Port "gre-c0a88e6f"
        Interface "gre-c0a88e6f"
type: gre
options: {df_default="true", in_key=flow, local_ip="192.168.142.35",
out_key=flow, remote_ip="192.168.142.111"}
    Port br-tun
        Interface br-tun
type: internal
    Port "gre-c0a88e6e"
        Interface "gre-c0a88e6e"
type: gre
options: {df_default="true", in_key=flow, local_ip="192.168.142.35",
out_key=flow, remote_ip="192.168.142.110"}
    Port patch-int
        Interface patch-int
type: patch
options: {peer=patch-tun}
    Port "gre-c0a88e22"
        Interface "gre-c0a88e22"
type: gre
options: {df_default="true", in_key=flow, local_ip="192.168.142.35",
out_key=flow, remote_ip="192.168.142.34"}
```

在 br-tun 内部，除了 patch-int 端口外，每一对 GRE Tunnel 在 br-tun 中都有一个对应的端口，当前计算节点与全部网络节点和其他计算节点之间都会建立一一对应的隧道端口。在 computer2 的 br-tun 中，端口 gre-c0a88e22 是由本地 IP 地址（192.168.142.35）到远端 IP 地址（computer1 节点 IP：192.168.142.34）之间的 GRE 隧道端口，除此之外，br-tun 中还有另外三个由本地 IP 地址分别至三个网络节点的 GRE 隧道端口。br-tun 使用节点上的标准路由表转发 GRE 封装后的数据包，因此不像 VLAN 网络，GRE 网络并不要求所有通信终端位于相同的二层 VLAN 网络中。

在步骤 3 的 br-int 网桥分析中，可以看到 computer2 计算节点的内部 VLAN Tag 是 1，

即在 GRE 模式下，由 patch-int 端口进入 br-tun 网桥的实例数据具有 VLAN ID=1 的内部 VLAN Tag，而 br-tun 的主要功能就是将内部 VLAN Tag 转换为 GRE Tunnel ID，之后通过对应的 GRE 端口转发到与对端节点互联的 GRE 隧道中。GRE Tunnel ID 可以通过 Neutron 命令行查看，如下：

```
[root@controller1-vm ~]$neutron net-show admin-net --fields provider:segmentation_id \
--fields provider:network_type
```

Field	Value
provider:network_type	gre
provider:segmentation_id	1

上述结果表明，GRE 网络中的 Tunnel ID 为 1。通过查看 br-tun 中的数据流规则，可以看到在 br-tun 内部节点中 VLAN ID 与 GRE Tunnel ID 之间的转换过程，转换过程由 br-tun 中的 OpenFlow 规则实现，OpenFlow 规则可以通过 ovs-ofctl 命令行工具查看^①，与 Tunnel ID 相关的 OpenFlow 规则可通过抓取 0x<provider:segmentation_id> 关键字来获取（segmentation_id 以 16 进制表示），如下：

```
[root@computer2 ~]# ovs-ofctl dump-flows br-tun|grep 0x1
cookie=0x0, duration=3417.765s, table=3, n_packets=79, n_bytes=9740, idle_
age=3138, priority=1,tun_id=0x1 actions=mod_vlan_vid:1,resubmit(,10)
cookie=0x0, duration=3417.854s, table=22, n_packets=15, n_bytes=1556, idle_
age=3224, dl_vlan=1 actions=strip_vlan,set_tunnel:0x1,output:2,output:5,outp
ut:4,output:3
```

在本例中，与 0x1 相关的 OpenFlow 规则有两条：第一条表明外部访问数据流的 GRE Tunnel ID 被转换成内部 VLAN ID（tun_id=0x1 actions=mod_vlan_vid:1），此处的 Tunnel ID=1 被转换成 VLAN ID=1（br-int 网桥的内部 VLAN Tag）；第二条规则表明具有内部 VLAN ID=1（由 br-int 网桥标记）的实例数据被转换成 Tunnel ID=1，同时将 VLAN ID 被剥离后并进行了 GRE 封装的数据发送到全部 GRE 输出端口，端口号分别为 2、3、4 和 5。OpenFlow 规则中看到的端口号可以通过 ovs-ofctl show 命令查看，如下：

```
[root@computer2 ~]# ovs-ofctl show br-tun
1(patch-int): addr:ce:eb:20:5e:41:26
config:      0
state:       0
speed: 0 Mbps now, 0 Mbps max
2(gre-c0a88e22): addr:1e:03:54:ad:67:25
config:      0
state:       0
speed: 0 Mbps now, 0 Mbps max
3(gre-c0a88e6e): addr:9e:05:9f:5d:4a:ba
```

① 此命令行工具帮助文档参考：<http://openvswitch.org/support/dist-docs/ovs-ofctl.8.txt>

```

config:      0
state:       0
speed: 0 Mbps now, 0 Mbps max
4(gre-c0a88e6f): addr:d6:71:6c:36:5d:52
config:      0
state:       0
speed: 0 Mbps now, 0 Mbps max
5(gre-c0a88e70): addr:d6:71:6c:36:5d:52
config:      0
state:       0
speed: 0 Mbps now, 0 Mbps max
LOCAL(br-tun): addr:de:4d:d1:33:bf:45
config:      PORT_DOWN
state:       LINK_DOWN
speed: 0 Mbps now, 0 Mbps max

```

5. 数据进入网络节点

计算节点中的实例数据经过 Linux 网桥、br-int 网桥和 br-tun (或 br-eth 网桥) 处理后, 通过 VLAN 网络或 GRE 隧道进入网络节点。在图 13-26 中, VLAN 模式下, 计算节点实例数据由物理网卡 eth1 发出, 经过二层 VLAN 物理网络后进入网络节点物理网卡 eth1, 而 eth1 属于物理连接网桥 br-eth1 的接口设备, 数据进入 br-eth1 后再由 phy-br-eth1 和 int-br-eth1 VETH 设备对进入 br-int 网桥。而在 GRE 模式下, 计算节点网络数据将直接进入网络节点 br-tun 隧道网桥, 经过 br-tun 处理后由 patch-int 和 patch-tun 进入集成网桥 br-int。下面对 VLAN 和 GRE 模式下网络节点接收数据后的处理方式进行分析。

(1) VLAN 模式

VLAN 网络模式中, 网络节点物理网卡 eth1 接收到带有外部 VLAN Tag 的数据, eth1 属于 OVS 物理连接网桥 br-eth1。同计算节点一样, br-eth1 与网络节点集成网桥 br-int 之间也有一对 VETH 设备互联, 即 phy-br-eth1 和 int-br-eth1, br-eth1 网桥将具有外部 VLAN ID 的数据转发至 phy-br-eth1 端口, 并由 VETH 设备对端 int-br-eth1 进入 br-int 网桥, br-int 网桥再将外部 VLAN ID 转换为节点内部 VLAN ID。VLAN 模式下, 网络节点集成网桥 br-int 与物理连接网桥 br-eth 的工作原理与计算节点类似, 网络节点内外部 VLAN ID 之间的转换可参考官网原理图, 如图 13-28 所示, 外部网络 VLAN101 和 VLAN102 数据帧通过网络节点 eth1 物理网卡进入网络节点, 并由 VETH 对 phy-br-eth1 和 int-br-eth1 进入集成网桥 br-int, br-int 的 int-br-eth1 端口接收到带有 VLAN101 和 VLAN102 的数据帧后, br-int 按照一定的转换规则将 VLAN101 和 VLAN102 分别转换为节点内部 VLAN Tag1 和 Tag2。与计算节点类似, br-int 中具有内部 VLAN Tag 的数据帧通过 int-br-eth1 和 phy-br-eth1 VETH 设备对进入物理连接网桥 br-eth1, br-eth1 的 phy-br-eth1 端口接收到带有内部 VLAN Tag 的数据帧后, 将内部 VLAN1 和 VLAN2 分别转换成外部 VLAN101 和 VLAN102, 并通过 br-eth1 网桥中的物理网络连接端口 eth1 进入外部网络交换机。

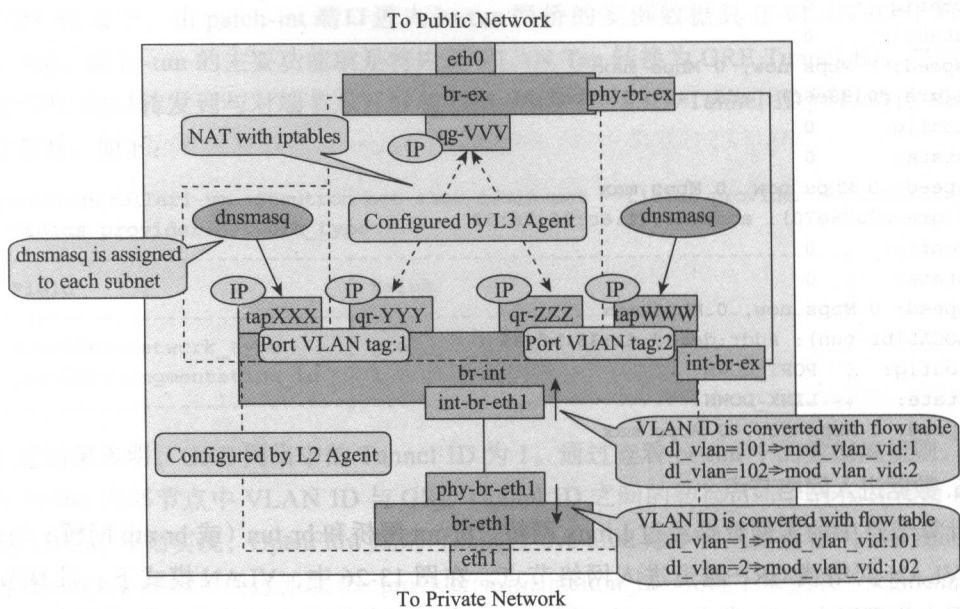


图 13-28 VLAN 模式下网络节点内外部 VLAN ID 转换

(2) GRE 模式

GRE 模式中，计算节点数据通过 GRE 隧道进入网络节点的隧道网桥 br-tun，之后由 br-tun 进行 GRE Tunnel ID 与节点内部 VLAN ID 之间的转换，转换后的数据由 VETH 设备对 patch-int 和 patch-tun 进入集成网桥 br-int。网络节点 GRE 隧道网桥 br-tun 与计算节点 br-tun 的工作原理类似，网络节点 br-tun 的 OpenFlow 规则如下：

```
[root@controller1-vm ~]# ovs-ofctl dump-flows br-tun
cookie=0x0, duration=3285.842s, table=3, n_packets=26, n_bytes=2626, idle_age=591, priority=1, tun_id=0x1 actions=mod_vlan_vid:1,resubmit(,10)
cookie=0x0, duration=840.516s, table=3, n_packets=435, n_bytes=23430, idle_age=1, priority=1, tun_id=0x2 actions=mod_vlan_vid:2,resubmit(,10)
cookie=0x0, duration=905.080s, table=22, n_packets=461, n_bytes=24834, idle_age=0, dl_vlan=2 actions=strip_vlan,set_tunnel:0x2,output:3,output:2,output:5,output:4
cookie=0x0, duration=3353.469s, table=22, n_packets=2, n_bytes=140, idle_age=3346, dl_vlan=1 actions=strip_vlan,set_tunnel:0x1,output:3,output:2,output:5,output:4
```

与计算节点 br-tun 的 OpenFlow 规则相比，网络节点 br-tun 多出了一个 Tunnel ID 0x2 和内部 VLAN ID2，这主要是因为 Neutron 的 L3 HA 高可用模式下，三个网络节点之间传输心跳信号需要使用特定的私有网络，该私有网络（默认为 169.254.192.0/18）使用的便是此 VLAN ID 和 Tunnel ID。与计算节点类似，网络节点 br-tun 实现 VLAN ID2/1 与 Tunnel ID 0x2/1 之间的转换，对于计算节点传输进来的 GRE 封装数据帧（或者其他网络节点传输进

来的 GRE 封装心跳信号数据帧), br-tun 将其 Tunnel ID 0x1 (或者 0x2) 转换为网络节点内部 VLAN ID 1 (或者 2) 之后由 patch-int 和 patch-tun VETH 设备对进入 br-int 网桥。而对于网络节点 br-tun 网桥传输进来的数据帧, br-int 将其内部 VLAN ID 1 (或者 2) 转换为 GRE Tunnel ID 0x1 (或者 0x2) 之后由 4 个 GRE 隧道端口发送出去, br-tun 内部的 GRE 隧道端口可以通过 ovs-ofctl 命令查看, 如下:

```
[root@controller3-vm ~]# ovs-ofctl show br-tun
1(patch-int): addr:7a:ab:39:e0:92:db
config:      0
state:       0
speed: 0 Mbps now, 0 Mbps max
2(gre-c0a88e22): addr:2e:0e:dd:a3:20:6b
config:      0
state:       0
speed: 0 Mbps now, 0 Mbps max
3(gre-c0a88e23): addr:aa:da:61:d5:91:7b
config:      0
state:       0
speed: 0 Mbps now, 0 Mbps max
4(gre-c0a88e6e): addr:06:41:db:32:78:43
config:      0
state:       0
speed: 0 Mbps now, 0 Mbps max
5(gre-c0a88e6f): addr:c6:54:fc:e0:74:9f
config:      0
state:       0
speed: 0 Mbps now, 0 Mbps max
```

6. 网络节点 br-int 网桥

网络节点 br-int 网桥与计算节点不同, 网络节点 br-int 网桥内部包含与 br-tun 通信的 VETH 设备 patch-tun (GRE 模式), 或与物理连接网桥 br-eth1 通信的 VETH 设备 int-br-eth1 外 (VLAN 模式), 还有与 DHCP 命名空间连接的 TAP 设备和与 L3 路由命名空间连接的 qr-xxx 接口设备, 以及和网络节点外部物理网桥 br-ex 连接的 VETH 设备 int-br-ex, 此外在 L3 HA 高可用模式下, 还存在与 L3 高可用路由连接的 ha-xxx 接口设备。在基于 Pacemaker 的 OpenStack 高可用集群中, 网络节点 br-int 网桥内部接口设备如下所示:

```
[root@controller3-vm ~]# ovs-vsctl show
Bridge br-int
    fail_mode: secure
    Port br-int
        Interface br-int
type: internal
    Port int-br-ex
        Interface int-br-ex
type: patch
options: {peer=phy-br-ex}
```

```

Port patch-tun
    Interface patch-tun
type: patch
options: {peer=patch-int}
    Port "ha-53c857b3-26"
tag: 2
        Interface "ha-53c857b3-26"
type: internal
    Port "qr-536e8eac-c5"
tag: 1
        Interface "qr-536e8eac-c5"
type: internal
    Port "tap071dc79d-3b"
tag: 1
        Interface "tap071dc79d-3b"
type: internal

```

在网络节点的 br-int 中，与 L3 HA 高可用心跳信号相关的 ha-53c857b3-26 端口内部 VLAN Tag 为 2，而与租户网络相关的 qr-536e8eac-c5 和 tap071dc79d-3b 端口内部 VLAN Tag 为 1，这也解释了在网络节点 br-tun 的 OpenFlow 规则中为什么存在 VLAN ID1 与 Tunnel ID 0x1 和 VLAN ID2 与 Tunnel ID 0x2 两条转换规则。注意在 L3 HA 模式中，网络节点的 ha-xxx 端口数目仅有一个，但是 qr-xxx 和 tapxxx 的数目则与 Neutron 网络中的租户网络数目相等，即 Neutron 网络中有 N 个租户网络，则网络节点 br-int 中应该存在 N 个 qr-xxx 和 tapxxx 端口。具体而言，每存在一个租户网络，便会创建一个 qdhcp 命名空间，因此在 br-int 上需要对应生成一个 TAP 设备与 qdhcp 连接，同时，如果多个租户网络均需要接入 L3 路由（qrouter 命名空间），则 br-int 上对应每个租户网络都会生成一个 qr-xxx 端口，以便租户网络与 qrouter 命名空间连接。两个租户网络分别接入两个 L3 路由命名空间的网络节点情况如图 13-29 所示，图中 br-int 网桥上共有两个 TAP 设备和两个 qr-xxx 设备，其中 br-int 网桥上的 tapAAA 和 qr-BBB 属于租户 VLAN102 网络（对应内部 VLAN Tag2），并分别接入到 qdhcp-dddd 和 qrouter-cccc 网络命名空间，而 tapXXX 和 qr-YYY 则属于租户 VLAN101 网络（对应 VLAN Tag1），并分别接入 qdhcp-aaaa 和 qrouter-bbbb 网络命名空间。

此外，还需注意到 br-int 网桥中的 int-br-ex 端口设备，此设备与网络节点外部网桥 br-ex 的 phy-br-ex 构成一对 VETH 设备，但是当内部租户网络与外部网络通过 L3 路由实现通信时，连接 br-int 网桥和 br-ex 网桥的 VETH 设备对 int-br-ex 和 phy-br-ex 中不会有任何数据帧通过。

7. L3 路由命名空间

在 Neutron 网络中，L3 路由是包含路由表和 iptables 规则集合的网络命名空间，并由 Neutron 项目中的 L3-agent 实现。由于 L3-agent 是有状态服务，因此其高可用不能通过简单的负载均衡形式实现，而 L3 路由的高可用设计也是整个 OpenStack 网络高可用的难点，目前主流的 L3 高可用方案主要有基于 VRRP 协议的 L3 HA 高可用方案和基于分布式

路由的 DVR 方案，关于 L3 高可用的设计方法可参考本书第 9 章中的内容。在图 13-26 中，br-int 网桥中的 qr-xxx 接口连接到对应的 L3 路由命名空间（L3 命名空间名称为 qrouter-<UUID>），同样在 L3 路由命名空间的内部接口中也会存在与 br-int 内部完全相同的 qr-xxx 接口。在 Linux 系统中，网络命名空间可通过 ip netns 命令查看，如下：

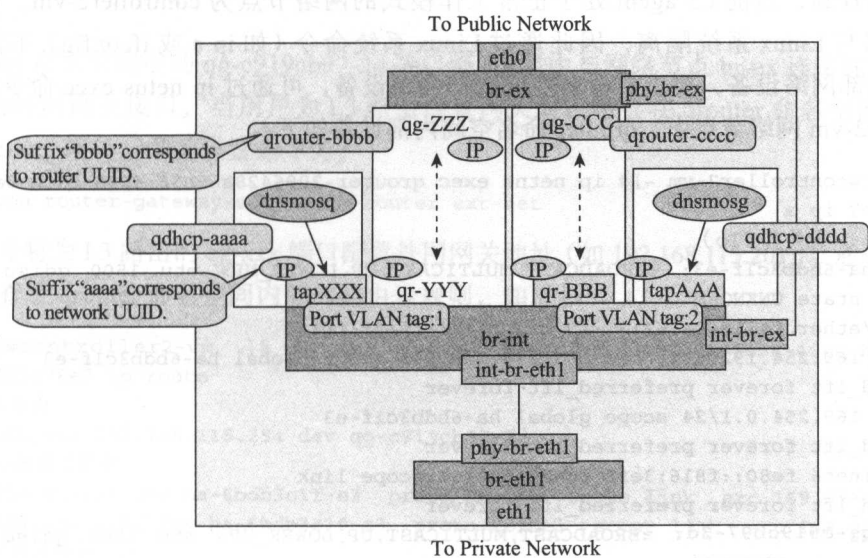


图 13-29 网络节点集成网桥与命名空间

```
[root@controller1-vm ~]# ip netns .
qrouter-39793e1d-719a-4a73-938a-fe04637a390c
qdhcp-cb63da5d-c59a-4c58-92a1-1070ea476e0d
```

其中，qdhcp-xxxx 是与租户网络对应的 DHCP 命名空间。在 L3 HA 高可用方案中，当用户创建高可用路由后，每个网络节点中均会出现具有相同 UUID 的 qrouter 命名空间。本例中，另外两个网络节点中的命名空间如下：

```
//controller2-vm命名空间
[root@controller2-vm ~]# ip netns
qrouter-39793e1d-719a-4a73-938a-fe04637a390c
qdhcp-cb63da5d-c59a-4c58-92a1-1070ea476e0d
//controller3-vm命名空间
[root@controller3-vm ~]# ip netns
qrouter-39793e1d-719a-4a73-938a-fe04637a390c
qdhcp-cb63da5d-c59a-4c58-92a1-1070ea476e0d
```

在 L3 HA 高可用方案中，L3 agent 工作在 A/P 高可用模式，因此三个网络节点中仅有一个 qrouter 命名空间为 Active 状态，其余两个应该为 standby 状态，如下：

```
[root@controller1-vm ~]$ neutron l3-agent-list-hosting-router admin-router
+-----+-----+-----+-----+-----+
| id                | host                | admin_state_up | alive | ha_state |
```

```

+-----+-----+-----+-----+-----+
| ddba16e7-e6b2-4276-a987-678f3b12e177 | controller2-vm | True | :- ) | active |
| d458cd1d-2a67-4f45-996a-8b5047c3d0c3 | controller3-vm | True | :- ) | standby|
| 5e851b76-cfb8-4bfe-8a83-1c8a036691bd | controller1-vm | True | :- ) | standby|
+-----+-----+-----+-----+-----+

```

可以看到，当前 L3 agent 处于正常工作模式的网络节点为 controller2-vm。由于命名空间内部与 Linux 系统隔离，因此通过 Linux 系统命令（如 ip a 或 ifconfig）不能查看命名空间内部网络设备，要查看命名空间内部网络设备，可通过 ip netns exec 命令实现。在 controller2-vm 网络节点中，qrouter 命名空间内部设备如下：

```

[root@controller2-vm ~]# ip netns exec qrouter-2096428a-6e58-4048-b3cb-0a5a255576
e7 ip a
..... (此处忽略lo)
38: ha-6bdb3c1f-e3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UNKNOWN
link/ether fa:16:3e:62:31:20 brd ff:ff:ff:ff:ff:ff
inet 169.254.192.6/18 brd 169.254.255.255 scope global ha-6bdb3c1f-e3
valid_lft forever preferred_lft forever
inet 169.254.0.1/24 scope global ha-6bdb3c1f-e3
valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe62:3120/64 scope link
valid_lft forever preferred_lft forever
39: qg-c919cb97-2d: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UNKNOWN
link/ether fa:16:3e:e6:d6:7d brd ff:ff:ff:ff:ff:ff
inet 192.168.115.201/24 scope global qg-c919cb97-2d
valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fee6:d67d/64 scope link nodad
valid_lft forever preferred_lft forever
40: qr-536e8eac-c5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UNKNOWN
link/ether fa:16:3e:0c:ad:e9 brd ff:ff:ff:ff:ff:ff
inet 192.128.1.1/24 scope global qr-536e8eac-c5
valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe0c:ade9/64 scope link nodad
valid_lft forever preferred_lft forever

```

其中，qr-536e8eac-c5 端口与网络节点 br-int 网桥连接，该端口在用户为 L3 路由添加租户子网接口时创建，设置命令为：

```
neutron router-interface-add admin-router admin-subnet
```

上述命令同时将为 qrouter 命名空间中的 qr-xxx 接口配置对应租户子网的网关 IP 地址（如 192.128.1.1）。用户为 L3 路由每增加一个接口，在 L3 路由命名空间和网络节点 br-int 中就会对应增加一个 qr-xxx 接口，而且 br-int 与 qrouter 内部 qr 接口名称完全一致，如 controller2-vm 节点内部的 br-int 中有如下接口：

```
[root@controller2-vm ~]# ovs-vsctl show
```

```

Bridge br-int
    fail_mode: secure
    Port "qr-536e8eac-c5"
tag: 1
    Interface "qr-536e8eac-c5"
type: internal
.....

```

qrouter 命名空间中的 qg-c919cb97-2d 接口将 L3 路由与网络节点 br-ex 连接在一起，是 L3 路由的外网网关接口，当用户为 L3 路由设置网关时，将会在 qrouter 命名空间和 br-ex 中创建 qg-xxx 端口，网关设置命令为：

```
neutron router-gateway-set admin-router ext-net
```

此命令将为 L3 路由的 qg-xxx 端口配置外网网关地址（如 192.168.115.201）。通过 ip route 命令可以查看 qrouter 命名空间内部的路由表规则，如下：

```

[root@controller2-vm ~]# ip netns exec qrouter-2096428a-6e58-4048-b3cb-0a5a
255576e7 ip route
//默认路由
default via 192.168.115.254 dev qg-c919cb97-2d
//HA心跳信号路由
169.254.0.0/24 dev ha-6bdb3c1f-e3 proto kernel scope link src 169.254.0.1
169.254.192.0/18 dev ha-6bdb3c1f-e3 proto kernel scope link src 169.254.192.6
//租户内网路由
192.128.1.0/24 dev qr-536e8eac-c5 proto kernel scope link src 192.128.1.1
//外网路由
192.168.115.0/24 dev qg-c919cb97-2d proto kernel scope link src 192.168.115.201

```

上述输出中，192.168.115.254 为外部物理网络网关，192.168.115.201 为 OpenStack 管理员设置的外网 ext-subnet 网关，而 192.128.1.1 为租户网络 admin-subnet 网关。L3 路由不仅负责将租户子网 IP 地址转换为外网网关地址（SNAT）以便租户网络可以访问外网，同时还负责将租户 Fixed IP 地址与 Floating IP 地址关联（DNAT）以便外网可以访问租户内部实例。本例中，当为 instance1 实例关联上 Floating IP 地址 192.168.115.202 后，qrouter 命名空间中的 NAT 路由表如下：

```

[root@controller2-vm ~]# ip netns exec qrouter-2096428a-6e58-4048-b3cb-0a5a
255576e7 \
iptables -t nat -S
.....
-A neutron-l3-agent-OUTPUT -d 192.168.115.202/32 -j DNAT --to-destination 192.
128.1.5
-A neutron-l3-agent-POSTROUTING ! -i qg-c919cb97-2d ! -o qg-c919cb97-2d -m
conntrack ! --ctstate DNAT -j ACCEPT
-A neutron-l3-agent-PREROUTING -d 169.254.169.254/32 -i qr-+ -p tcp -m tcp
--dport 80 -j REDIRECT --to-ports 9697
-A neutron-l3-agent-PREROUTING -d 192.168.115.202/32 -j DNAT --to-destination
192.128.1.5

```



```
-A neutron-l3-agent-float-snat -s 192.128.1.5/32 -j SNAT --to-source 192.168.115.202
-A neutron-l3-agent-snat -j neutron-l3-agent-float-snat
-A neutron-l3-agent-snat -o qg-c919cb97-2d -j SNAT --to-source 192.168.115.201
-A neutron-l3-agent-snat -m mark ! --mark 0x2 -m conntrack --ctstate DNAT -j SNAT --to-source 192.168.115.201
```

上述 NAT 路由表中，实现将实例 Fixed IP 地址 192.128.1.5 与 Floating IP 地址 192.168.115.202 进行 NAT 转换映射的 SNAT 与 DNAT 规则是：

```
-A neutron-l3-agent-OUTPUT -d 192.168.115.202/32 -j DNAT --to-destination 192.128.1.5
-A neutron-l3-agent-PREROUTING -d 192.168.115.202/32 -j DNAT --to-destination 192.128.1.5
-A neutron-l3-agent-float-snat -s 192.128.1.5/32 -j SNAT --to-source 192.168.115.202
```

同时 NAT 表中还有一条实现将租户内网（192.128.1.0/24）通过 SNAT 映射转换到外网网关 192.168.115.201 的 SNAT 规则（意味着没有 Floating IP 地址实例也可以访问外网），如下：

```
-A neutron-l3-agent-snat -o qg-c919cb97-2d -j SNAT --to-source 192.168.115.201
```

8. 外部网桥 br-ex

网络节点外部网桥 br-ex 与同计算节点通信的 br-eth1 网桥类似，两个网桥都至少包含一块物理网卡，只是同 br-ex 网桥连接的是 OpenStack 集群外部物理网络，即通常所说的 Public 网络。由计算节点进入网络节点的数据在经过 br-eth1 和 br-int 网桥后，由 br-int 网桥的 TAP 设备 qr-xxx 进入 L3 路由命名空间，通过 qrouter 命名空间的路由规则处理后，由外网网关接口 qg-xxx 进入外部网桥 br-ex。qrouter 命名空间内的 qg-xxx 与外网网桥 br-ex 内部的 qg-xxx 接口名称一致，如下：

```
Bridge br-ex
  Port phy-br-ex
    Interface phy-br-ex
  type: patch
  options: {peer=int-br-ex}
  Port br-ex
    Interface br-ex
  type: internal
  Port "eth0"
    Interface "eth0"
  Port "qg-c919cb97-2d"
    Interface "qg-c919cb97-2d"
  type: internal
```

其中的 qg-c919cb97-2d 端口与 qrouter 连接（qrouter 内部也有此端口），qrouter 中的数据经过路由处理，将由此端口进入 br-ex 网桥。

9. 进入外部网络

L3 路由根据 qrouter 命名空间中的 L3 路由表规则对数据进行路由后，将数据帧通过 qg-xxx 端口发送至 br-ex 网桥，br-ex 再将数据转发至内部接口 eth2（eth2 是与外网连接的物理网口），并由 eth2 进入物理网络中的下一个路由站点。

10. DHCP 命名空间

在 OpenStack 的 Neutron 网络中，与 L3 agent 类似，DHCP agent 也运行在网络命名空间中（DHCP 服务是一个运行在 Linux 系统中的 dnsmasq 进程实例），DHCP 命名空间以 qdhcp-<uuid> 形式命名。通常情况下，每个租户子网在网络节点对应一个 DHCP 命名空间，如果有多个租户网络，则对应多个 DHCP 命名空间，在基于 Pacemaker 的高可用 OpenStack 集群中，为了实现 Neutron 网络高可用，用户每创建一个租户子网网络，便会在三个网络节点中生成具有相同 UUID 的 qdhcp 命名空间，如下：

```
//此处仅有一个租户子网admin-subnet, ext-subnet为管理员创建的外网网络
[root@controller1-vm ~(keystone_admin)]$ neutron subnet-list
+-----+-----+-----+-----+
| id      | name      | cidr      | allocation_pools      |
+-----+-----+-----+-----+
| ...c7d  | ext-subnet | 192.168.115.0/24 | {"start": "192.168.115.200", "end": "192.168.115.250"} |
| ...0dd  | admin-subnet | 192.128.1.0/24 | {"start": "192.128.1.2", "end": "192.128.1.254"} |
+-----+-----+-----+-----+

//网络节点controller1-vm上的dhcp命名空间
[root@controller1-vm ~(keystone_admin)]$ ip netns
qdhcp-cb63da5d-c59a-4c58-92a1-1070ea476e0d
//网络节点controller2-vm上的dhcp命名空间
[root@controller2-vm ~]# ip netns
qdhcp-cb63da5d-c59a-4c58-92a1-1070ea476e0d
//网络节点controller3-vm上的dhcp命名空间
[root@controller3-vm ~]# ip netns
qdhcp-cb63da5d-c59a-4c58-92a1-1070ea476e0d
```

命名空间内部有自己的接口、路由表和 iptables 规则，要查看 qdhcp 命名空间内部的接口情况，可以通过 ip netns exec 命令实现，如下：

```
[root@controller1-vm ~]$ ip netns exec qdhcp-cb63da5d-c59a-4c58-92a1-1070ea476e0d ip a
..... (省略了lo接口)
18: tap8b597452-b7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UNKNOWN
link/ether fa:16:3e:49:d7:c5 brd ff:ff:ff:ff:ff:ff
inet 192.128.1.2/24 brd 192.128.1.255 scope global tap8b597452-b7
valid_lft forever preferred_lft forever
inet6 fe80::f816:3eff:fe49:d7c5/64 scope link
valid_lft forever preferred_lft forever
[root@controller2-vm ~]# ip netns exec qdhcp-cb63da5d-c59a-4c58-92a1-1070ea476e0d ip a
.....
```

```
18: tap0075f310-da: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
    UNKNOWN
link/ether fa:16:3e:68:6f:b8 brd ff:ff:ff:ff:ff:ff
inet 192.128.1.3/24 brd 192.128.1.255 scope global tap0075f310-da
    valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe68:6fb8/64 scope link
    valid_lft forever preferred_lft forever
[root@controller3-vm ~]# ip netns exec qdhcp-cb63da5d-c59a-4c58-92a1-1070ea
    476e0d ip a
.....
10: tap071dc79d-3b: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
    state UNKNOWN
link/ether fa:16:3e:b1:be:07 brd ff:ff:ff:ff:ff:ff
inet 192.128.1.4/24 brd 192.128.1.255 scope global tap071dc79d-3b
    valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:feb1:be07/64 scope link
    valid_lft forever preferred_lft forever
```

此外，在 OpenStack 集群中，通过 Neutron 命令行也可以发现针对某个特定的租户网络有多少个可用的 DHCP 服务器。在本书介绍的 OpenStack 高可用集群中，Neutron 网络服务由三个高可用网络节点组成，因此针对每一个租户网络应该有 3 个可用的 DHCP 服务器（可在配置文件中设置），如下：

```
[root@controller1-vm ~(keystone_admin)]$ neutron dhcp-agent-list-hosting-net admin-net
+-----+-----+-----+-----+
| id                | host                | admin_state_up | alive |
+-----+-----+-----+-----+
| 38752545-b6dc-413d-906f-0d2f2f8cb84a | controller3-vm | True            | :-)   |
| 6450d3ad-ffcf-4575-8ba1-9a13c7baf633 | controller1-vm | True            | :-)   |
| e228d0e1-245a-4c5f-917e-1809ae8814c8 | controller2-vm | True            | :-)   |
+-----+-----+-----+-----+
```

在 L3 HA 高可用网络的三个网络节点中，三个相同的 qdhcp 命名空间内部有不同的 TAP 设备，并且 TAP 设备上配置了与租户子网相同网段的 IP 地址，这三个 qdhcp 内部 TAP 设备的 IP 地址在租户子网 admin-subnet 创建时自动生成，如下：

```
[root@controller1-vm ~(keystone_admin)]$ neutron port-list --field id --field fixed_ips
+-----+-----+-----+-----+
| id                | fixed_ips            |
+-----+-----+-----+-----+
| 0075f310-da27-47dc-87fa-697ef32f8db3 | {"subnet_id": "...0dd", "ip_address": "192.128.1.3"}
| 071dc79d-3b57-43bb-acd2-dd6992b9cfbb | {"subnet_id": "...0dd", "ip_address": "192.128.1.4"}
| 8b597452-b759-419a-bf87-fe3f67bcc30a | {"subnet_id": "...0dd", "ip_address": "192.128.1.2"}
+-----+-----+-----+-----+
```

qdhcp 命名空间内部的 TAP 设备与网络节点 br-int 网桥内部的 TAP 设备连接，以

controller1-vm 中的 qdhcp 内部 TAP 设备 tap071dc79d-3b 为例，在 controller1-vm 网络节点内部，同样可以看到此 TAP 设备，如下：

```
[root@controller1-vm ~(keystone_admin)]$ ovs-vsctl show

Bridge br-int
    fail_mode: secure
    Port "qr-536e8eac-c5"
tag: 1
        Interface "qr-536e8eac-c5"
type: internal
Port "tap071dc79d-3b"
tag: 1
        Interface "tap071dc79d-3b"
type: internal
.....
```

同样，在另外两个网络节点的 br-int 网桥中也可以看到与 qdhcp 内部相同的 TAP 设备。上述网络节点 br-int 网桥内部，可以看到与 qdhcp 内部系统的 TAP 设备 tap071dc79d-3b，并且其 Tag 与租户子网端口 qr-536e8eac-c5 的 Tag 一致（均为 1），这说明与 qr-536e8eac-c5 相关的租户 IP 地址可以从与 tap071dc79d-3b 相关的 DHCP 服务中自动获取。

13.4.3 OpenStack Neutron 网络故障分析

Neutron 网络是 OpenStack 中相对较为复杂的部分，作为 OpenStack 新人，会经常遇到各种各样的网络问题。尽管随着 OpenStack 版本的不断更新，Neutron 项目也不断成熟，但是网络功能虚拟化（Network Function Virtualization, NFV）和软件定义网络（SoftwareDefineNetwork, SDN）仍然是 OpenStack 中 Neutron 网络项目较为复杂和难以理解的部分，也是各种 OpenStack 故障问题的“重灾区”。本节将对 Neutron 网络中常见的故障问题及其排除方法进行分析介绍，开源软件在不同的环境中总是出现不同的问题，因此 Neutron 网络问题没有固定的解决模式，更多的是娴熟的网络基础知识和触类旁通的思考方式，因此本节将按照以下几个通用步骤介绍 Neutron 网络故障排除方法，具体的理论知识还需参考前文知识点。

1. 基本环境检查

在很多时候，某个问题的出现并非因为多么深奥复杂的理论问题出错，而往往是由某些最基本的配置或简单的系统问题所导致，即所谓的“低级错误”。因此，在真正开始针对某个网络故障进行抽丝剥茧、寻根究底之前，检查最基本的变量设置和系统环境，很有可能消除你的懊恼并节约大量时间。在 OpenStack 的 Neutron 网络配置使用中，如果出现了各种诡异的网络问题，在依据网络实现原理对其进行跟踪分析之前，按照以下几个方面先进行基本问题的排查，或许对问题的排查会有极大帮助：

1）检查集群节点 NTP 时钟是否同步，各节点时钟同步是任何一个集群系统最基本的要求。

2) 检查节点文件系统空间使用情况, 没有足够的文件系统空间, 很可能导致各种服务进程无法启动, 或者启动后的服务也无法接受正常的访问, 尤其在开启 debug 模式时, 大量的日志或 dump 文件可能很快耗尽系统空间而引起各种服务异常。

3) 节点 CPU 使用情况, 在 Pacemaker 集群中, 当 CIB 进程在后台运行时, 将会消耗大量 CPU 资源, 尤其是在集群管理大量资源的情况下。对于集群资源而言, CPU 的过载使用很可能导致服务的启动或监控操作出现 Timeout 的错误。

4) 节点内存和硬盘等物理资源的使用情况, 当内存剩余量较低时, 计算节点将无法启动虚拟机, 控制节点 Pacemaker 集群运行缓慢甚至 Timeout, 当块存储后端磁盘空间不足时, Volume 创建也会报错。

5) 配置文件是否正常设置, 如 /etc/neutron/neutron.conf 或各种 Plugins 配置文件在未使用默认配置时, 自定义的参数是否正确。

6) 检查节点网络是否互通, 如计算节点与网络节点之间、各计算节点之间和各网络节点之间的物理网络是否可以正常 ping 通。

7) 检查 RabbitMQ 服务是否运行正常, OpenStack 中的大多数服务都依赖 RPC 通信机制, RabbitMQ 的异常或队列阻塞均会导致各种网络异常问题。

8) 检查数据库服务是否正常, Neutron 使用后端 MariaDB/MySQL 数据库存储和查询数据, 数据库服务的异常也会导致各种网络问题。

2. 故障问题定位

如果基本环境的检查都正常, 而网络问题仍然存在, 则需要进一步对问题进行定位分析。在网络出现异常问题时, 准确定位和缩小问题故障域对于后续问题解决分析十分关键。在网络路径中定位问题点最简单的方式便是通过 ping 来测试, 问题点的定位可按如下方式来实现:

1) 在实例中 ping 外部主机, 如 Google 提供的免费 DNS 服务器 IP 地址 8.8.8.8, 如果正常 ping 通则不存在任何网络问题。

2) 如果步骤 1 失败, 则尝试 ping 实例宿主计算节点的 IP 地址, 如果可以正常 ping 通, 则说明故障域应该在计算节点与计算节点网关之间。

3) 如果不能由实例内部 ping 通宿主计算节点主机 IP, 则说明网络故障域在实例和宿主计算节点之间, 可能的问题存在于将计算节点网卡与实例网卡连接在一起的各种网桥之间。

4) 再创建一个实例, 并测试两个实例彼此之间是否可以正常 ping 通, 如果可以正常互 ping, 则说明问题可能与计算节点防火墙相关。

3. 网络问题分析工具

在确定网络问题点之后, 下一步便是问题跟踪与分析, 在网络问题分析中, 选择合适的网络分析工具才会提取到极为关键的有效信息, 从而极大缩短故障解决时间, 常用的网

络分析工具有以下几种：

(1) ip a

在查看操作系统或网络命名空间内部的网络设备时，`ip a` (`ip addr` 的缩写) 命令非常有用。`ip a` 的输出中可以看到网络设备名称、IP 地址、设备是否 UP、MTU 值以及很多与网络相关的参数。

(2) route -n

`route -n` (或者 `ip route`) 命令显示路由表信息，根据路由表信息，可以发现当网络数据流向其他网络时，数据帧走的是哪一条路径。

(3) iptables -L

`iptables -L` 命令可以查看当前节点上有哪些防火墙规则，如果追踪到某条路径中的数据包突然消失或不能到达最终目的地，则很可能是由防火墙拒绝规则引起的。

(4) arp

`arp` 命令可以查看节点上的地址解析协议表 (`arp table`)，通过 `arp` 表信息，可以发现当前节点是否可以发现其他节点的 IP 地址。

(5) tcpdump

`tcpdump` 是网络故障排除经常使用的分析利器，`tcpdump` 工具由 `tcpdump` 软件包提供，用户仅需简单安装即可使用，`tcpdump` 命令行工具参数很多，基本用法就是监听某个网络端口，如 `tcpdump -i eth0`。

(6) ip netns

`ip netns` 主要用于网络命名空间操作，在针对 `qrouter` 或 `qdhcp` 命名空间的问题处理时，必须使用 `ip netns` 命令行工具，例如为了列出节点上的全部命名空间，可以使用如下命令：

```
ip netns list
```

此外，在 Linux 系统中可以使用的网络相关命令在命名空间中均可以使用，例如要查看 L3 路由命名空间内部的路由表，可以在命名空间中使用 `route -n` 命令，如下：

```
ip netns exec qrouter-UUID route -n
```

(7) ovs-vsctl/ovs-ofctl

`ovs-vsctl` 和 `ovs-ofctl` 命令是查看 OpenvSwitch 内部设备接口和 OpenFlow 规则的命令，如果在 Neutron 部署中使用到了 OpenvSwitch，则通过这两个命令可以进行调试和故障排除，需要注意的是，OpenvSwitch 提供的设备通过 Linux 系统命令是无法查看的。例如，要显示当前节点上由 OpenvSwitch 提供的网桥及其内部接口设备，使用如下命令：

```
ovs-vsctl show
```

要显示当前节点上特定网桥 (`br-int`) 的内部接口状态，使用如下命令：

```
ovs-ofctl show br-int
```


要显示当前节点上特定网桥 (br-int) 内部的全部 OpenFlow 规则, 使用如下命令:

```
ovs-ofctl dump-flows br-int
```

要显示当前节点上特定网桥 (br-tun) 内部的特定表中的 OpenFlow 规则, 使用如下命令

```
ovs-ofctl dump-flows br-tun table=21
```

(8) brctl

brctl 是 Linux 系统命令, 主要用于查看当前节点内部的 Linux 网桥情况, 当 Neutron 网络中使用到 Linux Bridge Agent 时 (通常是计算节点), 通过 brctl 命令可以查看当前 Linux 系统中的网桥及其接口情况, 显示系统中的全部网桥, 使用如下命令:

```
brctl show
```

显示 Linux 系统中特定网桥 (qbr152ac615-41) 信息, 使用如下命令:

```
brctl show qbr152ac615-41
```

下面针对新手较为常见的几个 Neutron 网络问题进行具体的步骤分析, 在 Neutron 社区中, 提及较多的问题不外乎实例不能与外网通信、实例获取不到 IP 地址以及端口状态异常等问题, 这几个问题的通用分析步骤如下。

问题 1 实例不能与外网通信

实例与外网通信通常由 L3 路由的 NAT 功能实现, 如果实例未关联外网浮动 IP 地址, 则将 L3 路由的外网网关地址作为 Source IP 地址, 并通过 SNAT 方式与外网通信, 如果已经为实例分配浮动 IP 地址, 则将实例浮动 IP 作为 Source IP 地址, 并以 SNAT 方式实现外网通信。当实例不能与外网通信时, 可能的原因并不唯一, 需要多方排查, 首先需要确保物理网络配置正确, 如交换机中是否允许 Neutron 配置的 VLAN ID 通过、计算节点与网络节点通信是否正常以及计算节点安全组规则是否允许 ICMP 包通过等, 要允许外网与实例之间进行 ssh 和 ping 操作, 需要设置正确安全组规则, 允许 ssh 和 ping 时, 正确安全组规则应该如下:

```
[root@controller1-vm ~]$ nova secgroup-list-rules default
```

IP Protocol	From Port	To Port	IP Range	Source Group
icmp	-1	-1	0.0.0.0/0	
tcp	22	22	0.0.0.0/0	

如果安全组规则正确, 在其他实例上检查是否可以 ping 通故障实例的私有 IP (Fixed IP), 如果租户私网内部 ping 失败, 则实例不可能与外网通信; 其次检查实例是否可以 ping 通路由网关, 如果失败则实例也不能与外网通信, 同样检查外网服务器是否可以 ping 通路由网关, 如果失败则实例肯定不能与外网通信。因为 L3 路由在此类故障中扮演关键作用,

因此可以进入 qrouter 命名空间中测试是否能与实例 Floating IP 地址通信，如下：

```
ip netns exec qrouter-88140ef2-4485-4ba6-9319-cac1c1c75c2b ping 192.168.115.204
```

其中，192.168.115.204 为实例 Floating IP 地址。由于实例与外网之间的通信需要经过计算节点和网络节点中的各种 Linux 网桥和 OVS 网桥，因此根据 13.4.2 节中的 Neutron 网络通信原理，使用 brctl show 和 ovs-vsctl show 命令检查 tapxxx、qvovxxx、qvbxxx、qbrxxx、qrxxx、qgxxx 以及 br-int、br-tun 等网络设备是否正常，并通过 ovs-ofctl dump-flows 命令检查 br-tun 内部 OpenFlow 规则是否正确对 VLAN ID 与 GRE Tunnel ID 进行了转换。另外，不排除代码错误或配置错误引起的故障，因此检查 Neutron 相关服务（尤其是 L3-agent）的 Log 文件是否发生错误，如下：

```
grep -E -i "error|trace" /var/log/neutron/l3-agent.log
```

从 VNC 控制台访问实例（如果 VNC 不能使用，也可以通过第三方 VNC 软件（如 VNC Viewer）来访问实例），在实例系统中查看路由规则并测试 ping 默认网关，如下：

```
route -n
ping default_gateway_ip
```

如果仍然不能找到问题原因，则可使用终极工具 tcpdump，tcpdump 可以双向捕捉经过网络设备的数据流。通常建议在网络数据流路径中的外网服务器、实例宿主计算节点和实例内部三个位置运行 tcpdump 以观察数据流情况，假设实例的固定 IP 和浮动 IP 分别为 192.128.1.5 和 192.168.115.204，路由网关为 192.168.115.203，外网服务器 IP 地址为 192.168.115.130，则在三个服务器上分别运行如下 tcpdump 命令：

```
tcpdump -i any -n -v 'icmp[icmptype] = icmp-echoreply or icmp[icmptype] = icmp-echo'
```

然后在实例中发起如下 ping 外网服务器 IP 地址的操作：

```
ping 192.168.115.130
```

如果实例（192.168.1.5）数据包能够成功到达外网服务器（192.168.115.130），则在计算节点上，将会看到 ping（192.168.1.5>192.168.115.130）和 ping 回复（192.168.115.130>192.168.1.5）成功通过，并且在计算节点上会看到重复的数据包，这是因为 tcpdump 捕捉了计算节点上的多个网络设备的数据包，如下：

```
[root@computer2 ~]# tcpdump -i any -n -v 'icmp[icmptype] = icmp-echoreply or icmp[icmptype] = icmp-echo'
tcpdump: listening on any, link-type LINUX_SLL (Linux cooked), capture size 65535 bytes
16:40:40.478351 IP (tos 0x0, ttl 64, id 8065, offset 0, flags [DF], proto ICMP (1), length 84)
  192.128.1.5 > 192.168.115.130: ICMP echo request, id 20225, seq 0, length 64
16:40:40.478369 IP (tos 0x0, ttl 64, id 8065, offset 0, flags [DF], proto ICMP (1), length 84)
  192.128.1.5 > 192.168.115.130: ICMP echo request, id 20225, seq 0, length 64
16:40:40.478372 IP (tos 0x0, ttl 64, id 8065, offset 0, flags [DF], proto ICMP (1), length 84)
  192.128.1.5 > 192.168.115.130: ICMP echo request, id 20225, seq 0, length 64
16:40:40.553183 IP (tos 0x0, ttl 63, id 35045, offset 0, flags [none], proto ICMP (1), length 84)
  192.168.115.130 > 192.128.1.5: ICMP echo reply, id 20225, seq 0, length 64
16:40:40.553204 IP (tos 0x0, ttl 63, id 35045, offset 0, flags [none], proto ICMP (1), length 84)
  192.168.115.130 > 192.128.1.5: ICMP echo reply, id 20225, seq 0, length 64
16:40:40.553214 IP (tos 0x0, ttl 63, id 35045, offset 0, flags [none], proto ICMP (1), length 84)
  192.168.115.130 > 192.128.1.5: ICMP echo reply, id 20225, seq 0, length 64
16:40:41.478908 IP (tos 0x0, ttl 64, id 8302, offset 0, flags [DF], proto ICMP (1), length 84)
```

在外网服务器上，将会看到服务器在接收 ping 请求（192.168.115.204>192.168.115.130），

并发送 ping 回复 (192.168.115.130>192.168.115.204), 如下:

```
[root@controller3 ~]# tcpdump -i any -n -v 'icmp[icmptype] = icmp-echoreply or icmp[icmptype] = icmp-echo'
tcpdump: listening on any, link-type LINUX_SLL (Linux cooked), capture size 65535 bytes
16:40:40.540398 IP (tos 0x0, ttl 63, id 8065, offset 0, flags [DF], proto ICMP (1), length 84)
  192.168.115.204 > 192.168.115.130: ICMP echo request, id 20225, seq 0, length 64
16:40:40.540415 IP (tos 0x0, ttl 63, id 8065, offset 0, flags [DF], proto ICMP (1), length 84)
  192.168.115.204 > 192.168.115.130: ICMP echo request, id 20225, seq 0, length 64
16:40:40.540494 IP (tos 0x0, ttl 63, id 8065, offset 0, flags [DF], proto ICMP (1), length 84)
  192.168.115.204 > 192.168.115.130: ICMP echo request, id 20225, seq 0, length 64
16:40:40.540524 IP (tos 0x0, ttl 64, id 35045, offset 0, flags [none], proto ICMP (1), length 84)
  192.168.115.130 > 192.168.115.204: ICMP echo reply, id 20225, seq 0, length 64
16:40:40.540746 IP (tos 0x0, ttl 64, id 35045, offset 0, flags [none], proto ICMP (1), length 84)
  192.168.115.130 > 192.168.115.204: ICMP echo reply, id 20225, seq 0, length 64
16:40:40.540751 IP (tos 0x0, ttl 64, id 35045, offset 0, flags [none], proto ICMP (1), length 84)
  192.168.115.130 > 192.168.115.204: ICMP echo reply, id 20225, seq 0, length 64
```

同时在实例上将会看到外网服务器发送的 ping 回复数据包 (192.168.115.130>192.128.1.5), 如下:

```
16:40:40.020974 IP (tos 0x0, ttl 61, id 8137, offset 0, flags [none], proto ICMP (1), length 84)
  192.168.115.130 > 192.128.1.5: ICMP echo reply, id 24895, seq 1, length 64
```

问题 2 实例获取不到 IP 地址

新创建的实例获取不到 IP 地址, 也是 Neutron 网络中较为常见的问题之一。实例不能自动获取 IP 地址, 通常与网络节点 DHCP 服务相关, 因此这类问题需要重点关注 DHCP 服务, 当然也不能排除实例镜像本身的问题, 要核实是否因为镜像问题而致使 IP 地址获取失败, 最简单的方式就是检查实例控制台输出, 如下:

```
openstack console log show <instance_name or UUID>
```

以 Cirros 镜像为例, 如果实例获取 IP 地址失败, 在控制台日志输出中会出现如下信息:

```
udhcpd (v1.17.2) started
Sending discover...
Sending discover...
Sending discover...
No lease, forking to background
starting DHCP for Ethernet interface eth0 [ [1;32mOK[0;39m ]
cloud-setup: checking http://169.254.169.254/2009-04-04/meta-data/instance-id
wget: can't connect to remote host (169.254.169.254): Network is
unreachable
```

而在正常获取 IP 地址的情况下, 实例控制台中关于 DHCP 的信息应该如下:

```
Starting network...
udhcpd (v1.20.1) started
Sending discover...
Sending select for 192.128.1.5...
Lease of 192.128.1.5 obtained, lease time 86400
cirros-ds 'net' up at 3.89
checking http://169.254.169.254/2009-04-04/instance-id
```

当确认实例已经正常引导启动后，之后便可继续判断问题所在。DHCP 问题很可能由于 dnsmasq 进程异常引起，在 Neutron 中，每个租户网络对应一个 DHCP 服务，而 DHCP 服务为 dnsmasq 进程实例，因此解决 dnsmasq 异常问题最简单的方式就是重启问题租户网络对应的 dnsmasq 进程，一旦重新启动了目标 dnsmasq 进程后，排除 dnsmasq 异常最简单的方式便是杀死网络节点上全部 dnsmasq 服务并重启 Neutron-dhcp-agent 服务，如下：

```
killall dnsmasq
systemctl restart neutron-dhcp-agent.service
//如果是pacemaker集群中，则重启dhcp资源
pcs resource restart neutron-dhcp-agent-clone
```

重启完成后，检查网络节点上应该出现的运行中的 dnsmasq 进程如下：

```
[root@controller3-vm ~]# ps aux | grep dnsmasq
nobody 1991 0.0 0.0 15544 836 ? S 18:08 0:00 /sbin/dnsmasq --conf-file=/
var/lib/libvirt/dnsmasq/default.conf --leasefile-ro --dhcp-script=/usr/
libexec/libvirt_leaseshelper
root 1993 0.0 0.0 15516 180 ? S 18:08 0:00 /sbin/dnsmasq --conf-file=/var/
lib/libvirt/dnsmasq/default.conf --leasefile-ro --dhcp-script=/usr/libexec/
libvirt_leaseshelper
nobody 22665 0.0 0.0 15544 828 ? S 18:13 0:00 dnsmasq --no-hosts --no-
resolv --strict-order --bind-interfaces --interface=tap0075f310-da --except-
interface=lo --pid-file=/var/lib/neutron/dhcp/cb63da5d-c59a-4c58-92a1-
1070ea476e0d/pid --dhcp-hostsfile=/var/lib/neutron/dhcp/cb63da5d-c59a-4c58-92a1-
1070ea476e0d/host --addn-hosts=/var/lib/neutron/dhcp/cb63da5d-c59a-4c58-92a1-
1070ea476e0d/addn_hosts --dhcp-optsfile=/var/lib/neutron/dhcp/cb63da5d-c59a-
4c58-92a1-1070ea476e0d/opts --leasefile-ro --dhcp-authoritative --dhcp-rang
e=set:tag0,192.128.1.0,static,86400s --dhcp-lease-max=256 --conf-file=/etc/
neutron/dnsmasq-neutron.conf --domain=openstacklocal
```

如果实例仍然没有获取到 IP 地址，则需检查 dnsmasq 进程是否监测到了来自实例的 DHCP 请求，在运行 dnsmasq 进程的网络节点上，运行如下命令检查 dnsmasq 输出：

```
grep -R -i DHCPDISCOVER /var/log/ -A 10 -B 10
```

如果 dnsmasq 进程监测到来自实例的请求并分配出一个 IP 地址，则应该有如下输出信息：

```
Nov 28 15:51:17 localhost dhclient[691]: DHCPDISCOVER on eth0 to 255.255.255.255 port
67 interval 6 (xid=0x140ae819)
Nov 28 15:51:18 localhost dhclient[691]: DHCPREQUEST on eth0 to 255.255.255.255 port
67 (xid=0x140ae819)
Nov 28 15:51:18 localhost dhclient[691]: DHCPOFFER from 192.168.142.254
Nov 28 15:51:18 localhost dhclient[691]: DHCPACK from 192.168.142.254 (xid=0x
140ae819)
Nov 28 15:51:18 localhost dracut-initqueue[562]: dhcp: BOND setting eth0
Nov 28 15:51:20 localhost dhclient[691]: bound to 192.168.142.33 -- renewal in 896
seconds.
```

如果不能看到 DHCPDISCOVER 信息，则表明问题应该在发出 DHCP 请求的实例至运

行 dnsmasq 进程的网络节点之间，如果可以看到上述输出，则说明实例 DHCP 请求已经正常到达 dnsmasq，但是 dnsmasq 没有做出正确的响应，此时很有可能是 dnsmasq 自身的问题，检查 neutron-dhcp-agent 日志文件，如下：

```
grep -E -i "error|trace" /var/log/neutron/dhcp-agent.log
```

此外，检查是否有 lease 文件，已经实例的 MAC 地址是否出现在 host 文件中，如下：

```
cat /var/lib/neutron/dhcp/<network_id>/host
```

如果感觉问题与 dnsmasq 无关，则使用 tcpdump 工具来测试接口上的网络数据包丢失的位置。DHCP 使用 UDP 协议，实例客户端使用 68 端口，DHCP 服务器端使用 67 端口，在使用 tcpdump 监听接口的同时，尝试重新创建一个实例，并系统性地监听端口，直至发现没有出现数据流的端口为止，例如监听 dnsmasq 节点 eth0 上的 67 端口，监听命令如下：

```
tcpdump -i eth0 -n port 67
```

问题 3 端口绑定失败

如果在创建实例时不能正常 boot，则检查是否在虚拟机、路由或 DHCP 上出现了端口绑定失败的问题，当虚拟机端口绑定失败时，其将会被标记为端口绑定失败，因此很好发现。而对于路由或 DHCP 端口的绑定失败，由于其端口是异步创建的，因此不是很容易发现，以路由端口创建为例，当创建一个路由并为其添加子网端口时，虽然返回消息表明添加成功，但是后台却正在进行端口创建过程，则创建过程很可能会失败。绑定失败的端口可通过如下方式查看：

```
[root@controller1-vm ~(keystone_admin)]$ neutron port-show \
1a73515a-f216-4c3d-adde-95a9a5fa5b6d -F "binding:vif_type" -F "binding:host_id"
+-----+-----+
| Field          | Value          |
+-----+-----+
| binding:host_id | computer2      |
| binding:vif_type | binding_failed |
+-----+-----+
```

如果通过 tail 命令跟踪创建实例计算节点上的 /var/log/nova/nova-compute.log 日志文件，将会看到如图 13-30 所示的 EEROR 级别日志信息，在图 13-30 中，traceback 语句“Nova Exception: Unexpected vif_type=binding_failed”中明确指出了异常原因为 binding_failed。

端口绑定失败很可能的原因有两个，一是当为路由添加新的子网或接口时 OVS agent 故障，可通过如下命令确认网络节点上的 OVS agent 运行情况：

```
neutron agent-list
```

此时将会看到 Open vSwitch agent 行的“alive”列下是“xxx”状态。此外，当 OVS agent 故障后，对应生成的 TAP 设备端口上没有 Tag 标记，可通过如下方式验证：

```
ovs-vsctl show | grep tap -A 3
```



```

ERROR nova.compute.manager [req-197db562-c8d2-4575-929c-6e644ee1424e - - - -] [instance: c5238b8d-536f-456e-9904-78e67a5c4f5d] Instance failed to spawn
TRACE nova.compute.manager [instance: ...] Traceback (most recent call last):
TRACE nova.compute.manager [instance: ...]   File "/usr/lib/python2.7/site-packages/nova/compute/manager.py", line 2442, in _build_resources
TRACE nova.compute.manager [instance: ...]     yield resources
TRACE nova.compute.manager [instance: ...]   File "/usr/lib/python2.7/site-packages/nova/compute/manager.py", line 2314, in _build_and_run_instance
TRACE nova.compute.manager [instance: ...]     block_device_info=block_device_info)
TRACE nova.compute.manager [instance: ...]   File "/usr/lib/python2.7/site-packages/nova/virt/libvirt/driver.py", line 2351, in spawn
TRACE nova.compute.manager [instance: ...]     write_to_disk=True)
TRACE nova.compute.manager [instance: ...]   File "/usr/lib/python2.7/site-packages/nova/virt/libvirt/driver.py", line 4172, in _get_guest_xml
TRACE nova.compute.manager [instance: ...]     context)
TRACE nova.compute.manager [instance: ...]   File "/usr/lib/python2.7/site-packages/nova/virt/libvirt/driver.py", line 4043, in _get_guest_config
TRACE nova.compute.manager [instance: ...]     flavor, virt_type)
TRACE nova.compute.manager [instance: ...]   File "/usr/lib/python2.7/site-packages/nova/virt/libvirt/vif.py", line 374, in get_config
TRACE nova.compute.manager [instance: ...]     _("Unexpected vif_type=%s") % vif_type)
TRACE nova.compute.manager [instance: ...]   NovaException: Unexpected vif_type=binding_failed
TRACE nova.compute.manager [instance: ...]
INFO nova.compute.manager [req-39d03ceb-a631-4b66-b77d-4cd9ee4ae01b] [instance: c5238b8d-536f-456e-9904-78e67a5c4f5d] Terminating instance
INFO nova.virt.libvirt.driver [-] [instance: c5238b8d-536f-456e-9904-78e67a5c4f5d] During wait destroy, instance disappeared.

```

图 13-30 端口绑定失败日志信息

此时，唯一的解决办法就是重新启动 OVS agent 并确认正常运行后，重新创建端口。

另外一个端口绑定失败可能的原因就是，Neutron 的 agent 或者 server 配置文件中出现错误的配置项，这种情况通常容易发生在未使用默认配置选项时。

13.5 OpenStack 日常管理与运维

13.5.1 OpenStack 日志设置管理与使用

在 OpenStack 中，大多数服务项目默认将日志写入 `/var/log/$service_name` 目录中，此处 `$service_name` 表示项目名称，如 nova、cinder 和 keystone 等，OpenStack 中各服务的运行节点及其日志存放位置如表 13-2 所示。

表 13-2 OpenStack 服务项目 Log 位置

节点类型	OpenStack 服务	Log 位置
Cloud controller	nova-*	/var/log/nova
Cloud controller	glance-*	/var/log/glance
Cloud controller	cinder-*	/var/log/cinder
Cloud controller	keystone-*	/var/log/keystone
Cloud controller	neutron-*	/var/log/neutron
Cloud controller	horizon	/var/log/apache2/
All nodes	misc (swift, dnsmasq)	/var/log/syslog
Compute nodes	libvirt	/var/log/libvirt/libvirtd.log

(续)

节点类型	OpenStack 服务	Log 位置
Compute nodes	nova-compute	/var/log/nova
Compute nodes	Console (boot up messages) for VM instances	/var/lib/nova/instances/instance-<instance id>/console.log
Block Storage nodes	cinder-volume	/var/log/cinder/cinder-volume.log

OpenStack 使用日志级别 (logging level) 来记录日志, 按照事件严重性递增, 这些日志级别分别是: DEBUG、INFO、AUDIT、WARNING、ERROR、CRITICAL 和 TRACE, 如果设置了某一级别, 则“严重性”高于此级别的信息都会被输出, 例如设置 logging level 为 DEBUG, 则所有级别的日志信息都会输出到日志文件中。由于 DEBUG 级别会输出大量日志, 因此如果不想保留无关的调试信息, 则可在配置文件中关闭 DEBUG 模式, 以 nova 为例, 在 /etc/nova/nova.conf 中进行如下设置以关闭 Nova 服务的 DEBUG 模式:

```
debug=false
```

Keystone 和 Horizon 的日志设置稍有不同, 对于 Keystone, 其日志 logging level 设置参数位于 /etc/keystone/logging.conf 文件的 logger_root 和 handler_file 配置段。而 Horizon 的 logging level 设置则位于 Python 文件 /etc/openstack_dashboard/local_settings.py 中, 因为 Horizon 是基于 Django 的 Web 应用程序, 因此其 logging 设置遵循 Django 日志框架约定。在 OpenStack 中, 要发现服务故障问题, 通常所做的第一步就是寻找日志文件中的 CRITICAL、TRACE 或 ERROR 信息, 带有 Python traceback 信息的 CRITICAL 日志如下:

```
2016-12-25 11:03:53 17409 CRITICAL cinder [-] Bad or unexpected response from the
storage volume backend API:volume group cinder-volumes doesn't exist
2016-12-25 11:03:53 17409 TRACE cinder Traceback (most recent call last):
2016-12-25 11:03:53 17409 TRACE cinder File "/usr/bin/cinder-volume", line 48, in
<module>
2016-12-25 11:03:53 17409 TRACE cinder service.wait()
2016-12-25 11:03:53 17409 TRACE cinder File "/usr/lib/python2.7/dist-packages/
cinder/service.py", line 422, in wait
2016-12-25 11:03:53 17409 TRACE cinder _launcher.wait()
2016-12-25 11:03:53 17409 TRACE cinder File "/usr/lib/python2.7/dist-packages/
cinder/service.py", line 127, in wait
2016-12-25 11:03:53 17409 TRACE cinder service.wait()
.....
```

上述 CRITICAL 日志信息来自 cinder-volume.log 文件, 根据 traceback 过程可以推断抛出异常的代码文件, 而 CRITICAL 行信息给出了异常问题的大致描述, 从上述信息中可以看出, 配置文件中指定的卷组 cinder-volumes 可能由于其他原因而不存在了, 或者创建卷组时指定的 VG 名称与配置文件不一致。以下日志信息为 ERROR 级别的异常日志:

```
2016-12-25 20:22:30 6619 ERROR nova.openstack.common.rpc.common [-] AMQP server
on localhost:5672 is unreachable: [Errno 111] ECONNREFUSED. Trying again in 21
```

seconds.

上述信息表示 Nova 服务与 RabbitMQ 失去联系，很可能的原因是由 RabbitMQ 服务故障导致。在 OpenStack 的运行维护中，最常见的问题就是实例不能正常运行，此时不得不在运行 Nova 各个子服务的控制节点和计算节点的 `/var/log/nova-*.log` 中搜寻相关的信息，此时通常的做法就是在各个 `nova-*.log` 中跟踪（使用 `grep`）故障实例的 UUID，例如在控制节点的 `nova-api.log`、`nova-scheduler.log` 以及计算节点的 `nova-compute.log` 中搜寻与故障实例 UUID 相关的 CRITICAL 或 ERROR 信息，如果在这些日志文件中均没有找到 CRITICAL 或 ERROR 级别信息，则日志文件中其他的 CRITICAL 或 ERROR 条目可能会提供有用信息。

如果认为源代码中提供的 logging 信息不够充分，则可以在相关的源代码文件中自定义 logging 信息，以 Nova 为例，其源代码文件位于 `/usr/lib/python2.7/dist-packages/nova` 目录。要增加 logging 语句，则应将下面的 Python 语句置于文件最前面（通常已经存在）：

```
from nova.openstack.common import log as logging
LOG = logging.getLogger(__name__)
```

之后，在需要添加 logging 信息的地方增加如下语句：

```
LOG.debug("This messages from warrior! ")
```

13.5.2 OpenStack 故障实例数据检查恢复

在 Nova 实例的运行维护中，有时会碰到实例正在运行但是却无法通过 SSH 访问，并且对任何命令均无响应，同时 VNC 控制台输出引导故障或 Kernel Panic 错误消息，此时很有可能是实例自身文件系统已经被损坏，并需要用户介入以修复故障的实例镜像文件系统。通常，要恢复实例文件或查看实例内容，首先需要使用 `qemu-nbd` 工具挂载实例的 disk 文件，实例 disk 文件位于 `/var/lib/nova/instances/<UUID>/` 目录。要访问实例 disk 文件，可按如下步骤操作：

- ❑ 使用 `virsh` 命令暂停 (`suspend`) 运行中的实例；
- ❑ 将 `qemu-nbd` 连接到实例 disk；
- ❑ 挂载 `qemu-nbd` 设备；
- ❑ 查看或修复实例镜像内容后卸载 `qemu-nbd` 设备；
- ❑ 断开 `qemu-nbd` 设备；
- ❑ 使用 `virsh` 命令重新启动 (`resume`) 实例。

通过 `qemu-nbd` 设备挂载实例 disk 文件后，便可如访问普通的目录文件结构一样访问实例镜像内容，但是最好不要在此时编辑或创建任何文件，因为此时的更改操作很可能改变实例镜像内部的访问控制列表（Access Control List, ACL），ACL 的改变过程可能导致实例无法引导启动。需要指出的是，在 Centos6/7 版本中，默认没有启用 NBD 模块，因此需要用户手工安装重新编译 Kernel 并启用 NBD 模块，对于最小安装的 Centos7 系统，可参照如

下脚本编译安装 NBD 模块：

```
#! /bin/bash
yum install kernel-devel kernel-headers
cd /tmp
//下载对应当前kernel版本的src.rpm包
wgethttp://buildlogs.cdn.centos.org/c7.1511.00/kernel/20151119220809\
/3.10.0-327.el7.x86_64/kernel-3.10.0-327.el7.src.rpm
//安装src.rpm包
rpm -ihv kernel-3.10.0-123.el7.src.rpm
cd ~/rpmbuild/SOURCES
//解压src.rpm包
tar Jxvf linux-3.10.0-123.el7.tar.xz -C /usr/src/kernels/
cd /usr/src/kernels/
mv $(uname -r) $(uname -r)-old
mv linux-3.10.0-123.el7 $(uname -r)
cd $(uname -r)
//重新编译kernel
make mrproper
cp ../$(uname -r)-old/Module.symvers ./
cp /boot/config-$(uname -r) /.config
make oldconfig
make prepare
make scripts
make CONFIG_BLK_DEV_NBD=m M=drivers/block
cp drivers/block/nbd.ko /lib/modules/$(uname -r)/kernel/drivers/block/
//加载nbd模块
depmod -a
modprobe nbd
```

验证是否正常加载 NBD 模块：

```
[root@computer1 ~]# lsmod |grep nbd
nbd                  17603  0
[root@computer1 ~]# ls -l /dev/nbd0
brw-rw---- 1 root disk 43, 0 Jan 17 10:15 /dev/nbd0
```

NBD 模块启用后，便可通过 NBD 设备挂载实例镜像进行查看。现假设当前租户正在运行两个实例 instance1 和 instance2，如下：

```
[root@controller1-vm ~(keystone_admin)]$ nova list
```

ID	Name	Status	Power State	Networks
48207d8f-adf8-412d-8974-5e53786a47d0	instance1	ACTIVE	Running	admin-net=192.128.1.5
390f20fa-81b6-4c70-821e-8319de6373e2	instance2	ACTIVE	Running	dmin-net=192.128.1.6

当前实例 instance2 不接受租户的任何操作指令，现在需要查看并修改实例 instance2 的

镜像内容。首先需要确认 instance2 位于哪台宿主计算节点，如下：

```
[root@controller1-vm rabbitmq(keystone_admin)]$ nova hypervisor-servers computer1
+-----+-----+-----+-----+
| ID                | Name                | Hypervisor ID      | Hypervisor Hostname|
+-----+-----+-----+-----+
| 390f20fa-81b6-4c70-821e-8319de6373e2 | instance-0000000c | 1                   | computer1          |
+-----+-----+-----+-----+
```

在 computer1 节点中，通过 virsh 命令确认虚拟机内部 ID，并通过 suspend 命令暂停 instance2，如下：

```
[root@computer1 ~]# virsh list
Id      Name                                State
-----
2       instance-0000000c                  running
[root@computer1 ~]# virsh suspend 2
Domain 2 suspended
```

将 qemu-nbd 设备连接到实例 disk 文件，如下：

```
[root@computer1 ~]# cd /var/lib/nova/instances/390f20fa-81b6-4c70-821e-8319de6373e2
[root@computer1 390f20fa-81b6-4c70-821e-8319de6373e2]# ls -l
total 2596
-rw-rw---- 1 qemu qemu 19123 Jan 17 09:08 console.log
-rw-r--r-- 1 qemu qemu 2686976 Jan 17 09:08 disk
-rw-r--r-- 1 nova nova 79 Jan 15 10:54 disk.info
-rw-r--r-- 1 nova nova 2626 Jan 17 09:07 libvirt.xml
[root@computer1 390f20fa-81b6-4c70-821e-8319de6373e2]# qemu-nbd -c /dev/nbd0 `pwd`/
disk
```

挂载 NBD 设备，qemu-nbd 测试将实例不同的分区挂载为不同的 NBD 设备，如实例中的 /dev/vda 是实例磁盘，而 /dev/vda1 是 root 分区，则 qemu-nbd 将 /dev/vda 挂载为 /dev/nbd0，而将 /dev/vda1 挂载为 /dev/nbd0p1。将 root 分区挂载到 /mnt 目录，如下：

```
[root@computer1 ~]# mount /dev/nbd0p1 /mnt
```

现在可以访问实例镜像文件，如下：

```
[root@computer1 ~]# ls -l /mnt
drwxrwxr-x 2 root root 3072 May 7 2015 bin
drwxr-xr-x 3 root root 1024 May 7 2015 boot
drwxr-xr-x 8 root root 3320 Jan 17 03:00 dev
drwxrwxr-x 13 root root 1024 Jan 17 03:02 etc
drwxrwxr-x 4 root root 1024 May 7 2015 home
.....
```

实例镜像内容查看完成后，卸载 NBD 设备并断开连接，如下：

```
[root@computer1 ~]# umount /mnt
[root@computer1 ~]# qemu-nbd -d /dev/nbd0
/dev/nbd0 disconnected
```

使用 virsh 的 resume 命令继续运行实例，如下：

```
[root@computer1 ~]# virsh list
Id      Name                                State
-----
2       instance-00000009                 running
3       instance-0000000c                 paused
```

```
[root@computer1 ~]# virsh resume 3
```

```
Domain 3 resumed
```

```
[root@computer1 ~]# virsh list
```

```
Id      Name                                State
-----
2       instance-00000009                 running
3       instance-0000000c                 running
```

13.5.3 OpenStack 故障计算节点实例恢复

在基于 Pacemaker 的 OpenStack 高可用集群中，计算节点故障将触发 Pacemaker 自动调用实例迁移代理程序，因此计算节点上的实例具有高可用性。本节再介绍一种快速批量恢复故障计算节点上全部实例的方式，在实际运行中，发生计算节点主板故障等严重性硬件故障时，该计算节点上的全部实例都将受到影响，如果短期内不能修复节点，同时计算节点没有被高可用集群软件（如 Pacemaker）保护，或者集群监控软件未能有效监控计算节点故障，在计算节点的 /var/lib/nova/instances 使用共享存储的情况下，可以将故障计算节点上的全部实例 relaunch 到其他节点。要在正常运行的计算节点上重新启动故障计算节点上的全部实例，可按如下方式操作：

从 nova 数据库中查询故障计算节点上的全部实例 UUID，如下：

```
[root@controller1-vm ~]#mysql -unova -pnova -e "use nova;select uuid from instances \
where host = 'computer1' and deleted = 0;"
+-----+
| uuid |
+-----+
| 48207d8f-adf8-412d-8974-5e53786a47d0 |
| 390f20fa-81b6-4c70-821e-8319de6373e2 |
+-----+
```

此处，假设 computer1 计算节点已经故障，从 nova 数据库中可以看到该节点上曾运行了两个实例。现在，更新 nova 数据库中的 instances 表，将宿主机为计算节点 computer1 的实例全部变更宿主机为 computer2，如下：

```
[root@controller1-vm ~]#mysql -unova -pnova -e "use nova;updateinstances set host \
='computer2'where host = 'computer1' and deleted = 0;"
//验证两个实例宿主机是否变更为computer2
[root@controller1-vm ~]#mysql -unova -pnova -e "use nova;select uuid from instances \
where host = 'computer2' and deleted = 0;"
```

```

+-----+
| uuid                                     |
+-----+
| 48207d8f-adf8-412d-8974-5e53786a47d0 |
| 390f20fa-81b6-4c70-821e-8319de6373e2 |
+-----+

```

如果使用的是 ML2 插件，则更新 neutron 数据库的 ml2_port_binding 和 ml2_port_binding_levels 表，将属于 computer1 主机的全部端口归属变更为 computer2，如下：

```

[root@controller1-vm ~]#mysql -uneutron -pneutron -e "use neutron;update ml2_
port_bindings\
set host = 'computer2'where host = 'computer1';"
[root@controller1-vm ~]# mysql -uneutron -pneutron -e "use neutron;update \
ml2_port_binding_levels set host = 'computer2'where host = 'computer1';"

```

现在，使用 nova 或 OpenStack 命令行重新启动上述两个受 computer1 故障影响的实例，同时在新的计算节点上生成实例虚拟机的 XML 文件，如下：

```

[root@controller1-vm ~]#openstack server reboot --hard instance1
[root@controller1-vm ~]#openstack server reboot --hard instance2

```

验证两个实例已经在新的计算节点 computer2 上重新运行，如下：

```

[root@controller1-vm ~]#nova hypervisor-servers computer2

```

```

+-----+-----+-----+-----+
| ID              | Name              | Hypervisor ID   | Hypervisor Hostname |
+-----+-----+-----+-----+
| 48207d8f-adf8-412d-8974-5e53786a47d0 | instance-00000009 | 1               | computer2           |
| 390f20fa-81b6-4c70-821e-8319de6373e2 | instance-0000000c | 1               | computer2           |
+-----+-----+-----+-----+

```

如果受影响的实例之前挂载有 Volumes，则按照如下步骤为重新启动的实例挂载 Volumes。首先，从数据库中生成实例和 volumes 的 UUID 列表，如下：

```

[root@controller1-vm ~]#mysql -uroot -proot -e"select nova.instances.uuid as
instance_uuid,
cinder.volumes.id as volume_uuid, cinder.volumes.status,
cinder.volumes.attach_status, cinder.volumes.mountpoint,
cinder.volumes.display_name from cinder.volumes
inner join nova.instances on cinder.volumes.instance_uuid=nova.instances.uuid
where nova.instances.host = 'c01.example.com';"

```

上述链表查询将会输出如下格式的临时表内容：

```

+-----+-----+-----+-----+-----+-----+
|instance_uuid|volume_uuid|status|attach_status|mountpoint|display_name|
+-----+-----+-----+-----+-----+-----+
|9b969a05     |1f0fbf36  |in-use|attached     |/dev/vdc  |test        |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

```


根据上述实例和 volumes 的 UUID 列表，使用 nova 或 openstack 命令行手工为实例 detach 并重新挂载 volumes，如下：

```
[root@controller1-vm ~]# openstack server remove volume <instance_uuid><volume_uuid>
[root@controller1-vm ~]#openstack server add volume <instance_uuid><volume_uuid> --device
/dev/vdx
```

13.5.4 OpenStack 实例间浮动 IP 地址管理

在使用 OpenStack 的 Neutron 网络时，可以使用 Provider 类型的网络或 Self-service 类型的网络，Self-service 类型具有很高的网络灵活性，租户自己可以创建网络、子网和路由等网络对象，并通过 L3 路由将租户网络与管理员定义的外部网络联通，同时使用 Floating IP 地址以 SNAT 和 DNAT 方式与外网双向通信。而在 Provider 类型网络中，L3 路由由物理设备实现，租户实例直接由云管理员创建的外网中获取 IP 地址，因此实例通过获取的 Fixed IP 地址即可实现与外网的通信，但是这种方式下实例 IP 地址将不具备类似 Self-service 网络中的 Floating IP 一样，可以随意 associate、disassociate 以及在不同的实例之间转移复用。不过，Neutron 提供了端口机制来实现 Provider 类型网络中的这种需求，现假设 OpenStack 环境中的实例通过 admin-net 网络可以获取被外网访问的 IPv4 地址，创建可被重用的 Neutron 端口过程如下。

在 admin-net 网络中创建 Neutron 端口，同时指定要分配给端口的 IP 地址（也可以由 Neutron 自动分配），并为端口指定一个方便记忆的名称，如下：

```
[root@controller1-vm ~(keystone_admin)]$ neutron port-create admin-net --fixed-ip \
subnet_id=ext-subnet,ip_address=192.168.115.219--name test-reused-port
Created a new port:
[root@controller1-vm ~(keystone_admin)]$ neutron port-create ext-net --name
reused-port Created a new port:
```

Field	Value
admin_state_up	True
allowed_address_pairs	
binding:host_id	
binding:profile	{}
binding:vif_details	{}
binding:vif_type	unbound
binding:vnictype	normal
device_id	
device_owner	
fixed_ips	{ "subnet_id": "...-aale3d5bec7d", "ip_address": "192.168.115.219" }
id	8f31b0ba-103b-4c4a-b675-f5e8432354e0
mac_address	fa:16:3e:15:b2:db
name	test-reused-port
network_id	beb90bc3-4ee1-46a4-8c6a-55063a357eb3

```
| security_groups | db1c21e1-87e0-4157-9a7d-97de72880f97 |
| status | DOWN |
| tenant_id | 5751ec6ea581413484cad0bed9c39bf4 |
+-----+-----+
```

创建实例时指定使用上述创建的 Neutron 端口，如下：

```
[root@controller1-vm ~(keystone_admin)]$ openstack server create --flavor 1\
--image cirros --key-name admin-key --nic port-id= 8f31b0ba-103b-4c4a-b675-f5e8432354e0\
--security-group default"instance-resued-port"
```

检查实例是否正确获取到与端口对应的 IP 地址，如下：

```
[root@controller1-vm ~(keystone_admin)]$ nova list
+-----+-----+-----+-----+-----+
| ID | Name | Status | Power State | Networks |
+-----+-----+-----+-----+-----+
| ...-dd0aae81c19b | instance-resued-port | ACTIVE | Running | ext-net=192.168.115.219 |
+-----+-----+-----+-----+-----+
```

可以看到，实例 instance-resued-port 已经获取到对应端口的 IP 地址（192.168.115.219）。现在，如果希望将此 IP 地址重新分配给其他实例使用（如果是 Floating IP，只需 disassociate 再重新 associate 即可），则可按如下方式操作：

找到与实例 IP 地址对应的端口，如下：

```
[root@controller1-vm ~(keystone_admin)]$ neutron port-list|grep 192.168.115.219
| 8f31b0ba-103b-4c4a-b675-f5e8432354e0 | reused-port | fa:16:3e:87:d0:3b | {"subnet_id": "3463dc5c-ff41-4ef8-a65a-aale3d5bec7d", "ip_address": "192.168.115.219"} |
```

使用 neutron port-update 命令将对应的端口从实例中移除，如下：

```
[root@controller1-vm ~(keystone_admin)]$ neutron port-update \
8f31b0ba-103b-4c4a-b675-f5e8432354e0 --device_id "" --device_owner ""
--binding:host_id ""
```

现在，使用 --nic port-id 选项即可重新将端口对应的 IP 地址（192.168.115.219）分配给新的实例使用，如下：

```
[root@controller1-vm ~(keystone_admin)]$ openstack server create --flavor 1\
--image cirros --key-name admin-key --nic port-id= 8f31b0ba-103b-4c4a-b675-f5e8432354e0\
--security-group default"new-instance-resued-port "
```

13.5.5 OpenStack 服务运行缓慢解决方案

在 OpenStack 集群运行维护中，经常会发现某个服务或多个服务响应变得缓慢，但是较长时间的等待后又可以得到正确的结果，此类情景通常较难以入手解决，而管理员首要应该是判断反应缓慢的影响范围，例如是单个服务反应缓慢，还是多个服务反应缓慢，如果问题可以缩小到某个特定的服务上，则可以通过重启该服务的形式临时性解决问题，

不过这也只是临时性的问题解决办法，却不是问题的根本原因。本节主要介绍一些针对运行缓慢问题的分析经验和思路，但并不是绝对正确或唯一的解决办法。

(1) Keystone 认证服务运行缓慢

Keystone 认证服务在使用较长时间后，可能会变得较为缓慢，而 Keystone 缓慢的原因很可能与日积月累的 Tokens 相关，这个问题可以在运行 Keystone 服务的控制节点上执行 `keystone-manage token_flush` 命令得到解决，

(2) Glance 镜像服务运行缓慢

访问镜像服务之前，必须经过 Keystone 的认证，因此 Glance 镜像服务的响应速度可能会受到 Keystone 响应缓慢的影响，但是 Glance 存储镜像文件的后端存储设备或文件系统出现故障，也可能是造成 Glance 自身响应缓慢的原因，如使用共享 NFS 文件系统存储镜像文件时，NFS 服务器因故障而无法访问。

(3) Cinder 块存储服务运行缓慢

OpenStack 块存储服务与 Glance 镜像服务类似，因此发现 Cinder 响应缓慢是首先检查与 Keystone 认证相关的服务，然后再去检查 Cinder 后端存储设备。此外，Cinder 和 Glance 服务都依赖 RPC 远程过程调用 SQL 后端数据库，因此在调试问题时不要忽视了 RabbitMQ 和 MariaDB/MySQL 数据库。

(4) Nova 计算服务运行缓慢

通常情况下，与 OpenStack 计算组件 Nova 相关的服务运行都相对较快，其响应速度通常与几个后端服务相关，即身份认证和授权服务 Keystone 与 RPC 远程过程调用 AMQP 服务 RabbitMQ。另外，与其他所有 OpenStack 服务一样，SQL 数据库也是必须考虑的部分。

(5) Neutron 网络服务运行缓慢

OpenStack 的 Neutron 网络响应缓慢的原因可能由其所依赖的服务引起，也可能跟物理或虚拟网络相关。例如，网络命名空间不存在或未正确连接到接口；被挂起或未正常运行的 DHCP 守护进程；物理网线被意外断开；交换机配置不正确等。要调试网络问题，首先应该核实各种物理网络相关的设备正常运行（如网线没问题，交换机配置没问题等），在确保物理网络设备正常运行后，检查核实 Neutron 相关的服务是否正常运行（`neutron-server`、`neutron-l3-agent` 和 `neutron-dhcp-agent` 等），最后检查 AMQP 消息队列和 SQL 后端数据库服务。

(6) AMQP 高级消息队列服务

无论使用哪种 AMQP Broker，如 RabbitMQ，总会在消息队列服务上碰到各种引起服务响应缓慢并直接影响到其他服务的 AQMP 问题。有时，消息队列会阻塞在消息队列服务中，并且不能被消息订阅者使用并清除，这种情况通常是由于消息队列服务或相关的 OpenStack 消息消费者服务被挂起或停止所引起，解决办法通常是重启消息队列服务或有问题的 OpenStack 服务来清除。此外，如果使用 RabbitMQ 高可用集群，则在 `/var/lib/rabbitmq` 目录中可能会生成大量的 `core` 文件，在 RabbitMQ 服务故障时，可以清除该目录

下的全部文件和目录，然后重启 RabbitMQ 服务。

(7) SQL 后端数据库服务

SQL 关系型数据库在 OpenStack 集群服务中起着非常关键的作用，不论使用的是 SQLite，还是 RDBMS（如 MySQL 或 MariaDB），数据库都承载了 OpenStack 大部分服务项目的配置和运行时状态数据存储服务。当使用文件存储作为后端时，大量的 SQLite 碎片文件可能致使 SQL 后端数据库变得缓慢，锁定或大量、长时间的查询通常导致 RDBMS 后端数据库响应缓慢，这种情况下，通常不建议立即 Kill 查询语句，而应该检查是否因为某些进程挂起或确实是有些查询请求需要长时间才能自动完成。此外，对于 MySQL/MariaDB 数据库，其默认的 `max_connections` 并不适合生产环境使用，在大量客户端请求访问数据库的情况下，连接数的限制将使得请求响应极为缓慢或者得不到数据库的响应。数据库的运行管理和调优本身是一门比较专业的技术，因此掌握或了解多种关系型数据库，如 MySQL/MariaDB、Oracle 或 DB2 等，对于确保 OpenStack 服务正常运行和优化调整可以起到很大的帮助。

13.5.6 OpenStack 配置文件及数据库备份

尽管 OpenStack 的高可用集群可以保证数据库和服务配置数据的多重保护，但是对于生产环境而言，多一份数据备份总会多一份安心。OpenStack 中的数据库备份可以有多重方式，根据需要备份数据的不同，可以是后端 SQL 数据库的备份、MongoDB 数据库的备份、服务配置参数的备份以及 Swift 对象存储和 Cinder 块存储生产数据的备份。本节主要介绍与 OpenStack 服务各项服务正常运行密切相关的配置文件备份和 SQL 关系型后端数据库备份，位于对象存储中的对象和块存储中的数据备份不在本节讨论范围，用户可以根据自己的备份环境对生产数据进行备份，如采用 NBU 或 TSM 等商业备份软件实现生产数据的备份。

1. 数据库备份恢复

在一般的 OpenStack 集群部署中，OpenStack 控制节点同时也作为 MySQL/MariaDB 数据库服务器。在基于 Pacemaker 的 OpenStack 高可用环境中，三个运行 MariaDB 服务的控制节点采用 Replication 形式实现数据同步，因此不论是在单机数据库服务器，或是多节点高可用环境中，只需在一个数据库服务器节点上进行数据库备份即可。在 OpenStack 安装部署中，OpenStack 的 Nova、Cinder、Neutron、Glance、Keystone 等服务均使用相同的数据库引擎，但是各自使用自己的数据库存储数据，由于这些数据库全位于同一个数据引擎中，因此可以很方便地实现各服务项目数据库的备份，例如要备份数据库引擎中的全部数据库，可通过如下命令实现：

```
mysqldump --opt --all-databases > openstack.backup
```

如果只需要备份单个数据库，如只备份 nova 数据库，则可通过如下命令实现：

```
mysqldump --opt nova > nova.backup
```

通常，OpenStack 的数据库并不会占用很大空间，因此备份频率可以适当提高，如每天备份。此时，便可通过 Linux 系统的 Crontab 来实现自动备份，实现 OpenStack 数据库自动备份的脚本可参考如下：

```
#!/bin/bash
backup_dir="/var/lib/backups/mysql"
filename="${backup_dir}/mysql-`hostname`-`eval date +%Y%m%d`.sql.gz"
//备份全部MySQL数据库，并压缩保存
/usr/bin/mysqldump --opt --all-databases | gzip > $filename
//删除保留一周以上的数据
find $backup_dir -ctime +7 -type f -delete
```

每天凌晨 1 点自动备份 OpenStack 数据库，Crontab 执行语句如下：

```
0 1 * * * /root/openstack_db_backup.sh >> /dev/null 2>&1
```

其中，openstack_db_backup.sh 为上述自动备份脚本文件名称（注意修改文件的可执行权限）。MySQL 数据库的恢复也非常简单，在恢复数据库之前，先确保相关的 OpenStack 服务已经停止，如要恢复 nova 数据库，则停止以下服务：

```
systemctl stop nova-api
systemctl stop nova-cert
systemctl stop nova-consoleauth
systemctl stop nova-novncproxy
systemctl stop nova-objectstore
systemctl stop nova-scheduler
```

然后，将需要恢复的 nova 数据库备份文件 nova.backup 直接恢复即可，如下：

```
mysql nova < nova.backup
```

恢复完成之后，重新启动上述 Nova 相关服务即可。如果是在 Pacemaker 集群中，则可以先直接恢复 nova 数据库，然后再重启 Nova 相关的资源，重启资源命令如下：

```
pcs resource restart nova-api-clone
```

因为 Nova 的其余资源依赖于 nova-api 资源，因此重启 nova-api 资源时，Pacemaker 会自动重启依赖资源。Nova 服务重启后，运行状态使用的便是最新恢复数据库中的数据。

2. 配置文件备份恢复

OpenStack 服务在启动运行时会读取配置文件信息，不同的配置文件参数将使得服务运行在不同的状态，错误的配置信息或配置文件丢失，对于 OpenStack 服务而言将是致命的。下面重点介绍 OpenStack 各服务组件需要备份的配置文件以及备份方式。

（1）计算服务

控制节点和计算节点中的 /etc/nova 目录是 Nova 计算服务需要备份的目录，/var/log/nova 是否需要备份应视情况而定，如果已经采用中心化的日志存储方式，则可以不用备份，

否则建议备份此目录。/var/lib/nova 是 Nova 服务中较为关键的目录，尤其是计算节点上的子目录 /var/lib/nova/instances，该目录存储了运行实例的 KVM 镜像，通常在需要维护实例的备份副本时才需要备份此目录，多数情况下无须备份此目录，需要注意的是，如果运行中的实例由备份中恢复而来，则运行时备份可能会导致实例不能正常启动。

（2）镜像服务

Glance 镜像服务的 /etc/glance 和 /var/log/glanace 是否需要备份与 Nova 服务的备份策略类似。/var/lib/glance 建议备份，尤其注意 /var/lib/glance/images 目录，如果使用文件后端存储 Glance 镜像，则此目录中存储了 Glance 镜像文件，因此最好对其备份。对 /var/lib/glance/images 目录的备份，通常可以有两种方式，一种就是将此目录挂载在基于 RAID 阵列磁盘的文件系统上，另一种是使用三方软件，如 rsync，将此目录的内容同步到其他服务器上备份，如下：

```
rsync -az --progress /var/lib/glance/images backup-server:/var/lib/glance/images/
```

（3）认证服务

/etc/keystone 和 /var/log/keystone 是否需要备份遵循其他组件的备份策略。至于 /var/lib/keystone 目录，虽然其不包含任何数据，但是以防万一，建议对其备份。

（4）块存储服务

/etc/cinder 和 /var/log/cinder 是否需要备份遵循其他组件的备份策略。/var/lib/cinder 也建议备份。

（5）网络服务

/etc/neutron 和 /var/log/neutron 是否需要备份遵循其他组件的备份策略。/var/lib/neutron 建议备份。

（6）计量统计服务

建议备份 /etc/ceilometer 目录。

（7）编排服务

备份 HOT 模板 yaml 文件，以及 Heat 配置文件 /etc/heat/。

配置文件的恢复非常简单，以恢复 Nova 服务的配置文件为例，假设 Nova 配置文件的备份存储在 /root/backup 目录中，则在恢复时只需将其拷贝覆盖 /etc/nova 目录即可，如下：

```
cp -a /root/backup/nova /etc/
```

配置文件恢复后，需要重新启动 Nova 相关的服务，如下：

```
#!/bin/bash
for i in openstack-nova-api openstack-nova-cert openstack-nova-consoleauth \
openstack-nova-novncproxy nova-scheduler
do
systemctl restart $i
done
```


13.6 本章小结

本章是对基于 Pacemaker 的 OpenStack 高可用集群运行维护的总结，在 Pacemaker 集群中，资源及其资源 OCF 代理脚本是集群管理资源的关键，本章对 OCF 脚本的语法、调试过程以及常见的日志跟踪方式进行了讲解，同时还对 Pacemaker 集群的进程日志系统进行了分析介绍，讲解了 Pacemaker 集群资源故障的常规排除方法。此外，本章还重点介绍了 OpenStack 高可用集群中的实例 HA 原理机制和计算节点故障监控脚本的实现方法，同时对监控脚本的执行过程进行了详细分析并对故障排除方法进行了介绍。在 OpenStack 运行维护中，Neutron 网络一直是较为复杂和出现问题最多的服务组件，本章对 Neutron 的网络路径进行了深入分析与讲解，并对 Neutron 网络中最为常见的故障问题及排除方法进行了介绍。最后，本章对 OpenStack 日常运维中最为常见的各种问题和操作进行了归纳总结。通过对本章的阅读，读者对 Pacemaker 和 OpenStack 集群的运行维护、实现原理和故障排除都会得到更深层次的理解。

Ceph 存储集群运维最佳实践

Ceph 与 OpenStack 整合部署几乎是目前开源云计算的标准选择，在 OpenStack 云计算的发展过程中，越来越多的用户选择使用 Ceph 作为 OpenStack 的后端存储，从而替换 OpenStack 社区的 Swift 对象存储和 Cinder 块存储项目。由于 Ceph 本身具有数据的自我愈合能力和高可用性，因此将 OpenStack 的数据存储分离至 Ceph 中，可以实现云计算中应用程序与数据存储的隔离，并降低 OpenStack 高可用集群的部署和维护难度，同时 Ceph 集群的统一存储和高性能特性使得 OpenStack 云计算具有更多样化的服务和更好的用户体验，因此无论是 OpenStack 社区，还是 Redhat、Maritans、Intel 以及诸多云计算公司都在大力支持 Ceph 存储，这也使得 Ceph 存储在 OpenStack 中的应用达到了空前高度。为了部署使用最优的 Ceph 存储集群，同时为了能够帮助用户快速掌握并解决 Ceph 存储集群在日常运行中遇到的故障问题，本章将从部署规划到运行维护对 Ceph 存储集群的最佳实践进行详细介绍。

14.1 Ceph 规划配置与性能调优

14.1.1 Ceph 硬件配置推荐

Ceph 存储集群通常部署在 X86 服务器上，因此与传统存储设备厂商的各种高端大容量存储阵列相比，对于很多中小客户而言，使用 Ceph 存储集群构建 PB 级别的数据存储池已不再遥不可及，Ceph 存储集群软件的开源高可用和 X86 服务器的低廉采购成本，使得 Ceph 在中小企业中被普遍接受。但是，在规划配置 Ceph 存储集群时，硬件服务器的选型配置对后期 Ceph 集群的性能起着决定性的作用，因此，做好前期硬件规划是保证 Ceph 集群正常、高效运行的基础，而 Ceph 集群的硬件选型通常需要综合各方面的因素，例如分布

式的 Ceph 进程（如 MON 进程和 OSD 进程）对服务器资源的消耗、客户端应用程序对服务器硬件资源的消耗以及存储磁盘的 IO 性能和服务器故障域等均是要考虑的因素。根据官方社区的建议和实际测试的结果，推荐将特定的 Ceph 进程部署在独立的服务器上，而访问 Ceph 存储的客户端程序（如 OpenStack、CloudStack 和 Zstack 等）部署在其他独立服务器上并通过高速数据网络访问 Ceph 集群。下面将从服务器的 CPU、内存、硬盘和网络等几个方面来描述如何选择 Ceph 集群服务器硬件配置。

1. CPU

在 Ceph 集群中，如果需要使用 CephFS 文件系统存储，则 Ceph 元数据服务器（MetaData Server，MDS）由于其负载动态重分配，因此其对服务器 CPU 较为敏感，所以作为 MDS 的服务器应该具备充裕的 CPU 计算能力（如 4 核或更多核 CPU 配置的服务器）。此外，Ceph OSD 运行 RADOS 服务，并负责根据 CRUSH Map 计算数据的存放位置、数据复制以及维护集群 Map 信息中指定的数据副本数，因此 OSD 服务器也应该具备一定的 CPU 处理能力（如双核 CPU）。假设服务器上共有 N 个物理 CPU，每个物理 CPU 有 K 核，每核时钟频率为 F GHz，Ceph 存储节点上有 n 个 OSD 进程，则 OSD 服务器 CPU 需求可按如下公式计算：

$$N * K * F / n \geq 1$$

相对 MDS 和 OSD 服务器而言，Ceph 监控服务 Monitor 仅维护集群 Map 主副本信息，其对 CPU 的依赖很小。在实际部署中，如果想要在运行 Ceph 进程的节点上运行其他服务，如在 Ceph 的 Monitor 节点上运行 KVM 虚拟机，则需要做额外的 CPU 消耗考虑。原则上，不建议在运行 Ceph 进程的主机上部署其他 CPU 密集型的应用程序。

2. 内存

Ceph 的 MDS 和 Monitor 服务需要快速的数据响应能力，因此必须为其提供足够的内存支持（每个进程至少 1GB 内存），而 OSD 进程在日常操作中无需大量内存（每个进程 500MB 内存即可），不过在 OSD 数据恢复过程中，其对内存消耗比较大（每个进程进行 1TB 数据的恢复需要约 1GB 内存），为了保证性能稳定，推荐为每个 OSD 进程预留多余的内存。

3. 数据存储

Ceph 中的数据存储配置需要谨慎规划，在考虑数据存储规划时，必须在巨额成本开销和存储集群的高性能之间做出权衡。在实际运行中，并行的系统操作和来自多个进程对单一驱动的并发读写有可能极大降低 Ceph 存储集群的性能。此外，对磁盘进行格式化的文件系统方式也是需要考虑的问题，btrfs 文件系统具有并行写日志和数据的能力，但是对生产系统而言其还不是很稳定，而 EXT4 和 XFS 虽然相对稳定，却不具备这种并行写数据的能力。在 XFS 和 EXT4 中，Ceph 必须将所有数据写入日志才会返回 ACK 信号，因此在日志和 OSD 性能之间进行权衡是十分重要的。

4. 机械硬盘

从成本上考虑,单盘容量越大则每 GB 容量的性价比越高,因此推荐使用大容量硬盘作为 OSD 磁盘,对于作为 Ceph OSD 的硬盘,其容量至少应为 1TB。此外,单盘存储空间越大,则每个 OSD 进程对内存的需求就越大,尤其是在 Ceph 集群进行数据重平衡 (Rebalance)、回填 (Backfill) 和恢复 (Recovery) 期间,常规而言,需要为 1TB 的存储空间预留 1GB 的内存。此外,不推荐通过分区形式在单个存储盘上运行多个 OSD,或者单盘上同时运行 OSD、MDS 和 Monitor 进程。

存储驱动器受限于寻道时间、访问时间、读写时间和总吞吐量,这些物理上的局限性影响着这个 Ceph 集群的性能,尤其在集群数据恢复期间。因此推荐将操作系统和应用软件安装在独立的存储驱动器上,另外每个驱动器上最多运行一个 OSD 守护进程。当然,Ceph 本身允许在每块硬盘驱动器上运行多个 OSD,但这会导致资源竞争并降低总体吞吐量,同时 Ceph 也允许把日志和对象数据存储在不同的驱动器上,但这会由于竞争关系而增加客户端写延时,因为 Ceph 必须先将数据写入日志盘,之后才返回客户端写成功并将日志数据同步到 OSD 对象数据盘中,虽然 btrfs 文件系统可以同时写入日志数据和对象数据,但是 XFS 和 EXT4 却不能。根据 Ceph 最佳实践,推荐采用单独的存储驱动运行操作系统、存储 OSD 对象数据和 OSD 日志数据。

5. SSD 固态硬盘

在 Ceph 集群中,提升存储性能的一种有效方式就是采用固态硬盘 SSD 来降低客户端随机访问时间和读延时,同时增加系统吞吐量。虽然与机械硬盘相比,SSD 的每 GB 存储空间成本通常要高出 10 倍以上,但是 SSD 的访问时间比机械硬盘至少快 100 倍。由于 SSD 没有移动机械部件,因此其不存在类似机械硬盘的局限性。而评价 SSD 性能的指标主要是顺序读写性能,例如在 SSD 盘为多个 OSD 存储多份日志文件时,顺序读写吞吐量为 400MB/s 的 SSD 性能要远高于 120MB/s 的 SSD 盘。SSD 盘有着较高的拥有成本,同时在 Ceph 集群中不同的使用方式可能带来不同的性能提升,因此在准备大规模投入使用 SSD 时,务必考虑清除如何使用 SSD 盘,并在使用前做好性能评估和测试。在 Ceph 集群中,SSD 最常见的使用方式就是用作 OSD 的日志盘,当 SSD 在作为日志盘使用时,还需要考虑如下几个重要的性能问题:

(1) 密集写

Ceph 日志记录通常与写密集型语句相关,因此在集群写数据时,至少要保证 SSD 日志盘的写性能等同于或高于 OSD 数据盘性能。由于市场上某些高性能的机械盘具有优于廉价 SSD 的写性能,因此这类廉价 SSD 盘在提升读访问性能的同时可能带来写延迟。所以在选择 SSD 时不要过于侧重低成本,过于廉价的 SSD 可能换来较大的 Ceph 性能的牺牲。

(2) 顺序写

当使用单个 SSD 盘存储多个 OSD 日志时,必须考虑此 SSD 盘的顺序写性能限制,因为此 SSD 盘可能面临多个 OSD 同时写日志的请求。

(3) 分区对齐

引起相同 SSD 盘存在不同性能的一个原因是分区对齐，大多数客户喜欢将 SSD 盘进行分区使用，但是却忽略了 SSD 盘的分区对齐问题，同分区对齐的 SSD 相比，未进行分区对齐的 SSD 在数据传输时要慢很多。

SSD 也可以作为数据对象的存储盘，但是这种方案的成本通常难以承受。在 Ceph 集群实际应用中，日志盘读写性能往往是 OSD 的性能瓶颈，因此将 OSD 日志存储在 SSD，即将 OSD 数据对象存储在机械盘上便可得到高性价比的 OSD 性能。对于 CephFS 文件系统，可以将 CephFS 的 Metadata 从文件内容中分离出来，并存储在基于 SSD 的 Pool 中，便可得到高性能的 CephFS 文件系统。

6. 网络

目前市面上大多数的机械硬盘都能达到 100MB/s 的吞吐量，为了能够处理存储节点上全部 OSD 硬盘的总吞吐，建议 Ceph 存储节点上至少部署两张千兆网卡，分别用于公网（前端）和集群网络（后端）。原则上集群网络不应接入公网，集群网络主要承载集群内部数据复制所产生的网络负载，同时位于内网中的集群网络可以防止 DDOS 攻击。DDOS 攻击会导致集群对 OSD 进行数据复制时 PG 无法回到 active+clean 的状态。如果条件允许，推荐在生产环境中部署万兆网络，假设在千兆网络中传输 1TB 数据需要 3 小时，则 3TB 数据需要 9 小时，相比之下，万兆网络内传输 1TB 仅需 10 分钟，而 3TB 仅需 1 小时。在 PB 级别的 Ceph 存储集群中，OSD 故障几乎是经常出现的问题，而此时网络带宽将直接影响到 PG 状态的恢复时效，例如在万兆网络中 OSD 故障后，PG 从 degrade 状态到 active+clean 状态的恢复时间要远小于千兆网络中的恢复时间。

7. 硬件配置参考

Ceph 存储集群可以运行在不同配置的硬件服务器上，对于小规模生产环境或开发测试环境，可以采用廉价和低配的服务器运行 Ceph，但是对于大规模高可用生产环境，最好在具有强大配置的服务器上运行 Ceph 集群。表 14-1 是 Ceph 官方网站推荐的运行 Ceph 的最小硬件资源配置要求。

表 14-1 Ceph 集群硬件资源最小推荐表

Ceph 进程	服务器资源名称	最小推荐值
Ceph-osd	CPU	1x 64-bit AMD-64 1x 32-bit ARM dual-core or better
	内存	~1GB for 1TB of storage per daemon
	存储卷	1x storage drive per daemon
	日志盘	1x SSD partition per daemon (optional)
	网络	2x 1GB Ethernet NICs

(续)

Ceph 进程	服务器资源名称	最小推荐值
Ceph-mon	CPU	1x 64-bit AMD-64 1x 32-bit ARM dual-core or better
	内存	1 GB per daemon
	磁盘空间	10 GB per daemon
	网络	2x 1GB Ethernet NICs
Ceph-mds	CPU	1x 64-bit AMD-64 1x 32-bit ARM dual-core or better
	内存	1 GB minimum per daemon
	磁盘空间	10 MB per daemon
	网络	2x 1GB Ethernet NICs

表 14-2 是 Ceph 官方社区提供的基于 Dell 服务器的 Ceph 生产环境配置参考，表中给出了可作为 Ceph OSD 存储节点的两种 Dell 型号服务器的配置，当然这并非标准的服务器配置（如系统盘应该由 Raid 盘构成），但是这至少是生产环境服务器最起码的配置。

表 14-2 生产环境服务器配置参考

服务器型号	服务器资源	最小推荐值
Dell PE R510	CPU	2x 64-bit quad-core Xeon CPUs
	内存	16 GB
	存储卷	8x 2TB drives.1 OS, 7 Storage
	客户端网络	2x 1GB Ethernet NICs
	OSD 网络	2x 1GB Ethernet NICs
	管理网络	2x 1GB Ethernet NICs
Dell PE R515	CPU	1x hex-core Opteron CPU
	内存	12x 3TB drives. Storage
	系统盘	1x 500GB drive. Operating System.
	存储卷	12x 3TB drives. Storage
	客户端网络	2x 1GB Ethernet NICs
	OSD 网络	2x 1GB Ethernet NICs
	管理网络	2x 1GB Ethernet NICs

14.1.2 Ceph 配置文件设置

Ceph 存储集群可以运行数千个 OSD，而为了实现数据复制，最小的 Ceph 集群也应该至少包含两个 OSD。不论 Ceph 集群大或小，都可以通过配置参数对其进行控制，而 Ceph

配置参数需要写入主机配置文件中，Ceph 启动时，相关进程会自动从配置文件中提取参数并实现对集群的控制。Ceph 提供了很多配置参数的默认值，用户可以在配置文件中对其进行覆写，此外还可以通过命令行工具对配置参数进行运行时修改。通常在启动 Ceph 集群后，会存在四种 Ceph 守护进程，即 `ceph-osd`、`ceph-mon`、`ceph-mds` 和 `ceph-radosgw`，其中 `ceph-osd` 和 `ceph-mon` 是必须存在的进程，而 `ceph-mds` 仅在启用 CephFS 文件系统功能时才存在，`ceph-radosgw` 仅在启用对象存储时才存在，当 Ceph 进程启动时便会加载主机配置文件，从而根据配置参数来运行 Ceph 集群。

在正式部署 Ceph 之前，建议选取适当的文件系统（Ceph OSDs 高度依赖底层文件系统的稳定性和性能）并对硬盘进行合理设置，例如生产环境中建议使用 XFS 或 EXT4 文件系统，而 XFS 对底层文件系统扩展属性（XATTR）有更好的支持，因此生产环境中首选 XFS 文件系统，尽管 BTRFS 文件系统是最为理想的选择，但是其稳定性仍需验证，建议在开发测试环境中使用。通常为了保证数据安全性，Ceph 在将数据写入硬盘驱动后才认为数据写入成功，因此在较老的 Linux 内核（2.6.33 以前版本）中，应该禁用硬盘的写缓存功能（如禁用 `hda` 硬盘写缓存，命令为 `hdparm -W 0 /dev/hda 0`），在新的内核版本中，并不存在这一问题。

当 Ceph 启动时，每个 Ceph 进程都会寻找提供了集群配置参数的配置文件（如默认配置文件 `/etc/ceph/ceph.conf`），如果是手工部署，则需用户自己创建配置文件，而使用工具部署时（如 `Ceph-deploy`、`Chef` 等）配置文件被自动生成，用户可以参考部署工具生成的配置文件。Ceph 的配置文件定义的内容包括：集群标志、验证设置、集群成员、主机名称、主机地址、秘钥路径、日志路径、数据路径、其他运行时选项。

1. Ceph 配置文件

Ceph 配置文件采用 ini 风格，配置行前端的“#”或“;”为注释符。配置文件既可以是对 Ceph 存储集群的全部进程进行配置，也可以仅针对特定类型的进程进行配置，要在同一个配置文件中同时配置多个 Ceph 进程，需要将配置选项放入不同的配置段中，`ceph.conf` 由以下几个配置段组成：

- `[global]`：在 `[global]` 配置段中的配置选项将影响全部 Ceph 存储集群进程。如授权方式配置便属于 `[global]` 全局配置段：

```
auth supported = cephx
```

- `[osd]`：在 `[osd]` 配置段中的配置选项将影响全部 Ceph 存储集群中的 `ceph-osd` 进程，同时覆盖 `[global]` 选项中同名的配置。如 OSD 日志大小配置便属于 `[osd]` 配置段：

```
osd journal size = 1000
```

- `[mon]`：在 `[mon]` 配置段中的配置选项将影响 Ceph 存储集群中的 `ceph-mon` 进程，同时覆盖 `[global]` 选项中同名的配置。如监控服务器的 IP 地址配置选项便属于 `[mon]` 配置段：

```
mon addr = 192.168.142.10:6789
```

- ❑ [mds]：在 [mds] 配置段中的配置选项将影响 Ceph 存储集群中的 ceph-mds 进程，同时覆盖 [global] 选项中同名的配置。如 MDS 主机名称选项配置便属于 [mds] 配置段：

```
host = mdsserver01
```

- ❑ [client]：在 [client] 配置段中的配置选项将影响全部 Ceph 客户端，如挂载的 Ceph 文件系统、RBD 块设备等。客户端日志设置便属于 [client] 配置段：

```
log file = /var/log/ceph/radosgw.log
```

在 [global] 中的全局设置将影响 Ceph 集群中的全部进程实例，因此在 [global] 中的配置选项是全部 Ceph 进程的通用配置。此外，[global] 中的配置可以通过两种方式进行覆盖，一种是在特定的进程类型（如 [osd]、[mon]、[mds] 和 [client]）配置段中进行配置修改，另一种是在特定的进程（如 [osd.1]）配置段中进行配置修改。在 Ceph 配置文件中，[osd]、[mon] 和 [mds] 称为进程类型配置段，如 OSD 类型进程、MON 类型进程和 MDS 类型进程，在进程类型配置段中的配置选项将影响全部该类型的进程实例，如 [osd] 中的配置将影响全部 OSD 类型进程实例（如 osd.1、osd.2、osd.3 等），因此如果需要对特定的进行实例（如 osd.1）进行控制，则需要在特定的进程实例配置段（如 [osd.1]）中进行配置选项的覆盖。Ceph 配置文件不同配置段影响范围如下：

```
[global]
//影响全部Ceph存储集群的进程
auth cluster required=cephx
auth service required=cephx
auth client required=cephx
[osd]
//影响全部OSD类型进程
osd journal size=1000
[osd.1]
//仅影响osd.1进程，覆盖 [osd] 和 [global] 配置段同名参数
.....
[mon.a]
//仅影响mon.a进程，覆盖 [mon] 和 [global] 配置段同名参数
.....
[mds.b]
//仅影响mds.b进程，覆盖 [mds] 和 [global] 配置段同名参数
.....
[client.radosgw.instance_name]
//仅影响client.radosgw.instance_name进程，覆盖 [client] 和 [global] 配置段同名参数
.....
```

为了简化 Ceph 存储集群的动态配置，Ceph 支持类似 bash 扩展的元变量。当在配置文件中使使用元变量时，Ceph 自动将元变量扩展为正确的具体变量值，Ceph 支持以下几种元变量：

- ❑ `$cluster`: 扩展为 ceph 存储集群的名称, 默认值为 `ceph`。当同时运行多个 Ceph 集群时, 使用此元变量将自动获取当前 Ceph 集群的名称。如 `/etc/ceph/$cluster.keyring` 默认情况下将被扩展为 `/etc/ceph/ceph.keyring`。
- ❑ `$type`: 扩展为 Ceph 进程类型名称, 可能的值分别为 `osd`、`mon` 和 `mds`, 例如在 `[osd]` 配置段中使用此元变量 `/var/lib/ceph/$type`, 则其将被自动扩展为 `/var/lib/ceph/osd`。
- ❑ `$id`: 扩展为进程标志符 (不是进程 ID), 如对于 OSD 类型进程 `osd.1`, 则其被扩展为 `1`, MON 类型进程 `mon.a`, 则其被扩展为 `a`。
- ❑ `$host`: 扩展为当前主机名称。
- ❑ `$name`: 扩展为 `$type.$id`。
- ❑ `$pid`: 扩展为进程 ID。

Ceph 运行用户对 Ceph 集群的 `ceph-osd`、`ceph-mon` 和 `ceph-mds` 进程进行运行时更改, 运行时更改对于实时增加 / 减少日志输出、启用 / 关闭调式模式以及运行时优化都是非常有用的。运行时配置更改的使用语法如下:

```
ceph tell {daemon-type}.{id or *} injectargs --{name} {value} \
[--{name} {value}]
```

其中, `{daemon-type}` 代表进程类型, 可能的值为 `osd`、`mon` 和 `mds`, `{id or *}` 意味着可以通过运行时设置更改特定类型进程中的全部进程实例 (*) 或者某个进程实例 (`id`)。例如要为 `osd` 类型进程 `osd.0` 增加调试日志, 则可执行如下命令:

```
ceph tell osd.0 injectargs --debug-osd 20 --debug-ms 1
```

在 Ceph 配置文件中, 可以在配置选项单词间使用空格, 而在命令行中, 相同配置选项的单词之间应该使用下划线或横线 (“`_`” 或 “`-`”), 例如在 `ceph.conf` 中的 `debug osd` 配置选项在命令行中应该为 `--debug-osd`。除了可以在运行时更改 Ceph 配置选项, 还可以在运行时查看特定进程所使用的配置参数值, 这对于 Ceph 调优是非常有用的。在 Ceph 调优时, 通常需要知道当前 Ceph 进程的配置参数, 然后再对配置参数进行运行时更改以便实现调优的目的。进程运行时配置参数的查看语法如下:

```
ceph daemon {daemon-type}.{id} config show | less
```

例如要查看 OSD 类型进程 `osd.0` 的运行时配置参数, 则可以执行如下命令:

```
ceph daemon osd.0 config show | less
```

2. Ceph 网络设置

网络配置是高性能 Ceph 存储集群建设的关键, Ceph 存储集群并不对客户端请求进行路由或分发, 客户端请求直接访问 Ceph OSD 守护进程, 同时 Ceph OSD 守护进程执行客户端数据复制请求, 而数据复制和其他集群内部通信数据势必增加 Ceph 存储集群的网络负载。默认情况下, Ceph 认为集群内部仅存在单个网络, 即 “公网”, 但是用户可以为 Ceph

集群指定第二个网络，即“集群”网络，集群网络的设置可为 Ceph 集群带来极大的性能提升。因此，在进行 Ceph 部署规划时，推荐为 Ceph 集群设置一个公网（Public Network）和一个集群网络（Cluster Network），如图 14-1 所示。

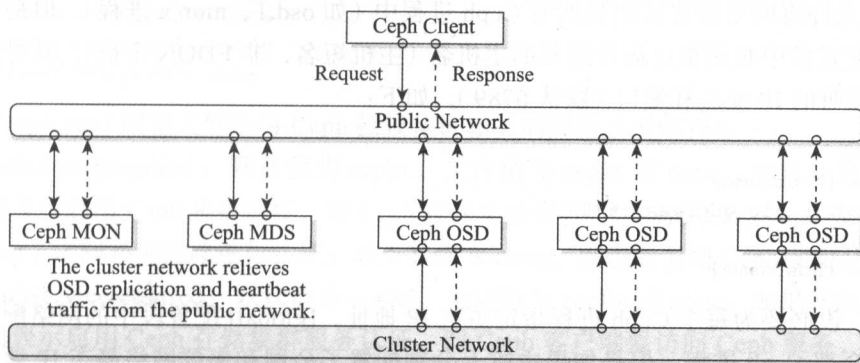


图 14-1 Ceph 网络

要实现图 14-1 中的网络，每个 Ceph 节点至少需要两张千兆网卡。推荐使用双网络的原因，可以从存储集群的性能和安全性上理解：

（1）性能

Ceph OSD 守护进程执行客户端的数据复制请求，如果 Ceph 集群仅共享一个公网，则当 Ceph 集群内部不断进行数据复制时，不同存储节点上 OSD 守护进程之间的数据复制网络负载必然增加，而 Ceph 客户端和 Ceph 存储集群之间的网络负载必然受到影响，从而容易出现延时或性能问题。此外，如果在 Ceph 仅有的公网上进行 Ceph 存储集群内部的数据恢复和再平衡过程，也会出现客户端访问延时和性能问题。因此，从集群性能上考虑，应该将 Ceph 集群内部网络负载迁移至集群网络，而公网仅负责客户端请求。

（2）安全

虽然多数时候网络环境是安全并受法律保护的，但是不排除少数恶意分子发起的 DDOS 攻击，如果 Ceph 存储集群仅有一个公网，则 OSD 守护进程之间的网络数据在受到干扰后，集群的 PG 状态将再也不能回到 active+clean 状态，因而客户端也没法读写 Ceph 集群数据。而防止类似攻击造成 Ceph 停止访问的方案之一就是增加一个完全独立的集群网络，使其与公网完全隔离。当然，也可以使用消息签名机制阻止欺骗性的攻击。

在 Ceph 的配置文件中，Ceph 网络配置通常添加至 [global] 全局配置段，虽然单网络（公网）情况下 Ceph 集群也能正常运行，但是在生产环境中，推荐使用公有网络和集群网络隔离并用的双网络形式。在网络定义时，不要混淆二者的用途，公网主要指 Ceph 客户端访问存储集群的网络，而集群网络指 Ceph 内部进行数据复制、恢复、心跳通信和数据再平衡时使用的内部网络，同时公网与集群网络之间应该完全隔离，即客户端不应该通过公网访问到集群网络。公有网络和集群网络在 Ceph 配置文件中的定义如下：

```
[global]
.....
public network = {public-network/netmask}
cluster network = {cluster-network/netmask}
```

Ceph 允许将网络配置应用到所有 Ceph 进程中（如 `osd.1`、`mon.a` 进程），但是在对应的进程实例配置段中必须指定运行进程的主机名（主机短名，非 FDQN 全称），同时需要指定监控进程监听的 IP 地址和端口（默认 6789），如下：

```
[mon.a]
host = {hostname}
mon addr = {ip-address}:6789
[osd.0]
host = {hostname}
```

通常，没必要为每个 Ceph 进程指定主机 IP 地址，`[global]` 配置段中的网络配置将会自动应用到全部 Ceph 进程，但是如果主机上分别配置了公网和集群网络静态 IP 地址，则可以为 Ceph 进程指定公网和集群网络主机 IP 地址，如下：

```
[osd.0]
public addr = {host-public-ip-address}
cluster addr = {host-cluster-ip-address}
```

3. Ceph 授权设置

Ceph 默认启用 Cephx 安全认证协议，尽管加密认证过程会带来一定的计算成本，但是为了防止类似“中间人（man-in-the-middle）”的攻击，在 Ceph 的 Client 和 Server 之间进行身份认证是十分必要的。当然，如果确认 Ceph 客户端与服务器之间的访问通信十分安全，则也可以关闭 Cephx 认证。Cephx 的配置过程根据用户部署 Ceph 的不同而不同；如果采用 Ceph-deploy 工具进行部署，则 Ceph-deploy 将自动配置 Cephx，用户无须做过多的设置；如果采用其他工具（如 Chef、Puppet）部署 Ceph，则必须手工配置 Cephx。

当使用 Ceph-deploy 部署 Ceph 时，用户无须手工创建 `client.admin` 用户及其秘钥文件，也无须手工实现 Ceph 监控进程的启动初始化过程。当用户在管理节点上执行 `ceph-deploy new {initial-monitor(s)}` 命令时，Ceph 将自动创建 `client.admin` 用户及其秘钥文件，同时生成一个 Ceph 配置文件 `ceph.conf`，而 Ceph 配置文件中的 `[global]` 配置段将会包含如下认证授权设置语句：

```
authclusterrequired = cephx
authservicerequired = cephx
authclientrequired = cephx
```

当用户执行 `ceph-deploy mon create-initial` 命令时，Ceph 将启动 Monitor 初始化过程，同时提取 `client.admin` 用户的秘钥文件 `ceph.client.admin.keyring`。当用户执行 `ceph-deploy admin {node-name}` 命令时，Ceph 配置文件 `ceph.conf` 和秘钥文件 `ceph.client.admin.keyring` 将被复制到目标节点 `/etc/ceph` 目录，从而在目标节点中即可通过 `root` 用户管理 Ceph 集群。

如果使用 Chef 和 Puppet 等工具部署 Ceph，则必须手工实现 Cephx 的认证过程，具体实现可参考 Ceph 官方网站^①。如果用户打算关闭 Cephx 认证，则只需在 Ceph 配置文件中做如下设置并重启 Ceph 集群即可：

```
authclusterrequired = none
authservicerequired = none
authclientrequired = none
```

关于 ceph.conf 配置文件中与 Ceph 授权认证相关的配置选项解释如下：

- ❑ `authclusterrequired`：默认值为 `cephx`，允许值为 `cephx` 或 `none`。如果其值为 `cephx`，则表示启用 Ceph 集群认证，即 Ceph 存储集群进程（如 `ceph-osd`、`ceph-mon` 和 `ceph-mds`）之间必须彼此进行认证；如果其值为 `none`，则表示关闭 Ceph 集群认证。
- ❑ `authservicerequired`：默认值为 `cephx`，允许值为 `cephx` 或 `none`。如果其值为 `cephx`，则表示启用 Ceph 存储集群服务认证，即 Ceph 客户端要访问 Ceph 服务，就必须与 Ceph 存储集群之间进行认证；如果其值为 `none`，则表示关闭 Ceph 客户端与 Ceph 存储集群服务之间的认证。
- ❑ `authclientrequired`：默认值为 `cephx`，允许值为 `cephx` 或 `none`。如果其值为 `cephx`，则表示启用 Ceph 存储集群客户端认证，即 Ceph 客户端要对 Ceph 存储集群进行认证；如果其值为 `none`，则表示关闭 Ceph 客户端对 Ceph 存储集群的认证。
- ❑ `cephx require signatures`：是否启用 Ceph 消息签名机制，默认值为 `false`，如果其值为 `true`，则表明 Ceph 客户端与 Ceph 存储集群之间的通信消息需要签名认证，同时 Ceph 存储集群内部进程之间的通信也需要签名认证。
- ❑ `cephx cluster require signatures`：是否启用 Ceph 存储集群签名机制，默认值为 `false`，如果其值为 `true`，则表明 Ceph 存储集群进程之间的通信消息需要签名认证。
- ❑ `cephx service require signatures`：是否启用 Ceph 客户端签名认证机制，默认值为 `false`，如果其值为 `true`，则表明 Ceph 客户端与 Ceph 存储集群之间的通信信息需要签名机制。
- ❑ `cephx sign messages`：默认值为 `true`，如果 Ceph 版本支持签名机制，则 Ceph 将为所有消息进行签名，签名后的消息可以阻止欺骗性攻击，因此建议此选项使用默认值。

4. Ceph 监控设置

Ceph Monitor 是 Ceph 存储集群中极为重要的部分，每个 Ceph 存储集群至少需要一个 Monitor 进程实例，用户可以实时添加或移除 Monitor，通常情况下，Ceph 存储集群中应该运行三个 Monitor 进程实例。Ceph 的 Monitor 维护着 Ceph 存储集群的 Cluster Map 副本信息，Cluster Map 存储了 Ceph 存储集群的内部结构信息，Ceph 客户端在读写 Ceph 存储集

① <http://docs.ceph.com/docs/master/install/manual-deployment/#monitor-bootstrapping>

群时，必须先连接到 Ceph Monitor 进程并获取 Ceph 的 Cluster Map 信息，从而确定 Ceph 全部 Monitor 进程、OSD 进程和 MDS 在 Ceph 存储集群中的位置。当 Ceph 客户端获取到 Cluster Map 后，结合当前集群的 Cluster Map 和 CRUSH 算法，Ceph 客户端便可计算出任何对象在 Ceph 存储集群中的存储位置，进而 Ceph 客户端可以直接与特定的 Ceph OSD 进程交互以读写存储数据，而 Ceph 客户端这种直接访问方式，正是 Ceph 存储集群具有极高扩展性和高性能的主要原因。

在 Ceph 存储集群中，Cluster Map 是获取当前 Ceph 集群状态最为重要的工具，Cluster Map 由 Monitor Map、OSD Map、MDS Map、PG Map 组成，Cluster Map 实时跟踪并记录很多 Ceph 存储集群中的重要信息，如 Ceph 存储集群中有哪些进程、Ceph 集群中哪些进程处于 up 或 down 状态、PG 是否为 active/inactive/clean 或者其他状态，以及能够反映集群当前状态的其他细节（如集群总共有存储空间、已经使用的存储空间和剩余存储空间等）。当 Ceph 集群状态出现较大变化时（如 OSD 进程故障、PG 进入 degrade 状态等），Cluster Map 也会随之实时更新以反映当前集群状态。此外，Ceph Monitor 进程也维护集群以前的历史状态信息，Monitor Map、OSD Map、MDS Map、PG Map 各自维护着自己的历史 Map 版本，通常将这些历史版本称为“epoch”。

Ceph Monitor 进程的配置选项由 Ceph 配置文件 ceph.conf 实现，如果需要对全部 Ceph Monitors 进程实现控制，则在 [mon] 配置段中设置选项；如果仅对特定 Monitor 进程实例（如 mon.a）进行配置，则在特定的进程实例配置段 [mon.a] 中设置选项。对 Ceph Monitor 进程而言，仅需包含主机名和监控 IP 地址的最小配置，Ceph Monitor 进程便可正常启动，由三个 Monitor 服务器组成的 Ceph Monitor 进程最小配置如下：

```
[mon]
mon host = hostname1,hostname2,hostname3
mon addr = 10.0.0.10:6789,10.0.0.11:6789,10.0.0.12:6789
```

此外，也可以将针对特定 Monitor 进程（如 mon.a）的配置拆分到特定的 [mon.a] 配置段，此时的配置选项仅针对 mon.a 进程，如下：

```
[mon.a]
host = hostname1
mon addr = 10.0.0.10:6789
```

在生产环境中，建议至少部署三个 Monitor 节点以便实现 Ceph Monitor 的高可用，如果想要部署更多的 Ceph Monitor 节点，则可以指定某几个必须成为集群成员的 Monitor 节点作为初始 Monitor 节点，以便建立集群的 Quorum 仲裁机制，同时减小集群启动时间，初始 Monitor 成员的设置过程如下：

```
[mon]
mon initial members = a,b,c
```

其中，a、b 和 c 表示初始 Monitor 节点的名称，初始节点数目通常为奇数以便满足 Quorum

要求。Ceph 为 Monitor 进程提供了默认的数据存储路径 (mon data), Ceph Monitor 以 Key/Value 键值对形式存储数据, 并遵循数据库 ACID 原则, 因此 Ceph Monitor 的数据存储具有强一致性, 通常并不推荐用户更改 Ceph Monitor 的默认数据存储路径, Ceph Monitor 的数据存储路径配置如下:

```
[mon]
mon data = /var/lib/ceph/mon/$cluster-$id
```

与 Ceph Monitor 相关的另外两个重要配置选项是与存储空间相关的 mon osd full ratio 和 mon osd nearfull ratio。mon osd full ratio 用于设置 Ceph 存储集群容量使用阈值, 例如当 mon osd full ratio 值为 0.95 或 95% 时, 即 Ceph 存储集群剩余空间不到总容量的 5%, 则 Monitor 认为 Ceph 存储集群空间已被耗尽, 此时为了保证数据安全性, Ceph 不会接受任何客户端的读写请求。而 mon osd nearfull ratio 用于设置 Ceph 存储集群可使用存储空间的告警值, 例如当 mon osd nearfull ratio 值为 0.85 或 85% 时, 即 Ceph 存储集群剩余空间不到总容量的 15%, 则 Monitor 认为容量已到警戒值, 并向管理员发出容量告警信息, 此时管理员必须重视, 否则当已用空间达到 mon osd full ratio 时, 所有 Ceph 客户端都无法访问 Ceph 集群, 在使用 Ceph 作为存储后端的 OpenStack 集群中, 典型的现象就是虚拟机没反应。mon osd full ratio 的默认值为 95%, mon osd nearfull ratio 默认值为 85%, 这两个选项通常位于 [global] 配置段, 如下:

```
[global]
mon osd full ratio = .80
mon osd nearfull ratio = .70
```

当运行多个 Monitor 节点时, 不同 Monitor 节点之间维护的 Cluster Map 可能存在差异, 此时需要以最新的 Cluster Map 为准, 因此相对落后的 Monitor 节点需要向拥有最新 Cluster Map 的节点同步信息, 即 Ceph 监视器之间的数据同步, 数据同步过程也可以通过配置选项进行控制。此外, 与 Ceph Monitor 相关的配置选项还有很多, 表 14-3 罗列了与 Monitor 相关且较为常见的配置选项, 实际应用中可以根据用户具体需求对这些配置选项进行修改。

表 14-3 Ceph Monitor 进程配置选项

配置选项	默认值	选项说明
mon sync trim timeout	30s	监视器数据同步完成后集群修整超时时间
mon sync heartbeat timeout	30s	监视器数据同步心跳超时时间
mon sync heartbeat interval	5s	监视器数据同步心跳间隔时间
mon sync timeout	30s	监视器数据同步超时时间
mon sync max retries	5	监视器数据同步最大尝试次数
mon sync max payload size	1045676	监视器数据同步负载最大尺寸
mon min osdmap epochs	500	监视器保留的历史 OSD Map 最大数目
mon max pgmap epochs	500	监视器保留的历史 PG Map 最大数目

(续)

配置选项	默认值	选项说明
mon max log epochs	500	监视器保留的 epoch 最大日志数目
mon clock drift allowed	0.05s	多个监视器之间允许存在几秒的时钟偏移
mon timecheck interval	300s	时钟漂移检查间隔
mon client hunt interval	3s	客户端每隔几秒连接一次监视器直到连接成功
mon client ping interval	10s	客户端间隔几秒 ping 一次监视器
mon max osd	10000	集群允许的 OSD 数目
mon sync fs threshold	5	写入多少个对象后开始于文件系统同步
mon subscribe interval	300s	监视器同步数据时间间隔

5. Ceph OSD 设置

Ceph OSD 进程可以在 Ceph 配置文件 ceph.conf 中进行配置，OSD 进程使用默认值和最小配置选项也能正常运行，最小配置选项仅需在 [osd] 配置段中设置 osd journal size 和 host 两个配置项即可。Ceph OSD 进程在 Ceph 存储集群内部按照整数由 0 开始编号，如 osd.0、osd.1、osd.2、osd.3 等，在对 OSD 进程进行配置时，既可以在 [osd] 配置段控制全部 OSD 进程，也可以在特定的 OSD 配置段（如 [osd.0]）仅控制 osd.0 进程。一个最简单的 OSD 进程配置选项如下：

```
[osd]
osd journal size = 1024
[osd.0]
host = osd-host-a
[osd.1]
host = osd-host-b
```

其中，“osd journal size = 1024”告知全部 OSD 进程（osd.0 和 osd.1）日志尺寸大小为 1GB，而“host = osd-host-a”仅告知 osd.0 进程主机名称为 osd-host-a，同样“host = osd-host-b”仅告知 osd.1 进程主机名称为 osd-host-b。在 Ceph 的 OSD 进程常规设置中，osd data、osd journal 和 osd journal size 是比较重要的设置，通常不建议更改 osd data 的默认值，而 osd journal 通常建议设置为 SSD 分区块设备，如果使用机械硬盘或文件路径存储 OSD 日志，则不建议与 OSD 进程共享同一块硬盘。OSD 日志大小可按如下公式计算：

$$\text{osd journal size} = \{2 * (\text{expected throughput} * \text{filestore max sync interval})\}$$

其中，“expected throughput”是磁盘吞吐和网络吞吐中的最小值，而 filestore max sync interval 表示 Ceph 将数据由日志盘同步至 OSD 中的最大时间间隔，间隔越大意味着日志盘需要存储的日志数据越多，因而 osd journal size 的值就越大。这三个配置参数的默认值如下：

```
[osd]
```

```
osd data = /var/lib/ceph/osd/$cluster-$id
osd journal = /var/lib/ceph/osd/$cluster-$id/journal
osd journal size = 5120
```

在 Ceph 集群中，除了为数据对象创建多个副本外，Ceph 还通过对 PG 执行 scrub 操作来保证数据的完整性。Ceph 的 scrub 操作类似于在对象存储层对数据进行 fsck 检查修复操作，Ceph 为每个 PG 上的全部存储对象生成对象目录，并将主对象与副本对象进行对比以确保没有任何对象丢失或损坏。默认每天进行的轻度 scrub 操作仅检查对象大小和属性，而默认每周进行的深度 scrub 读取全部数据并进行 checksums 校验以保证数据完整性。Ceph 的 scrub 操作对于保证数据完整性是十分重要的，但是 scrub 操作对集群性能有很大影响，因此 Ceph 提供了很多配置选项以控制 scrub 对性能的影响，如表 14-4 所示。

表 14-4 Ceph OSD 进程 Scrub 操作配置项

配置选项	默认值	选项说明
osd max scrubs	1	每个 OSD 进程所允许的并行 scrub 操作数目，值越大对集群性能影响越大
osd scrub begin hour	0	允许执行 scrub 操作的时间段起始值，取值范围为 0~24
osd scrub end hour	24	允许执行 scrub 操作的时间段结束值，取值范围为 0~24，执行 scrub 操作的时间段必须在 begin 与 end 之间
osd scrub during recovery	True	在集群数据恢复期间允许执行 scrub 操作
osd scrub thread timeout	60s	等待 scrub 线程超时的时间
osd scrub load threshold	0.5	最大负载，当 Ceph 集群检测到系统负载大于此值时，scrub 操作不被允许执行
osd scrub min interval	60*60*24s	Ceph 存储集群负载较低时，对 OSD 进程执行 scrub 操作的最小时间间隔，默认为每天一次
osd scrub max interval	7*60*60*24s	不论 Ceph 集群负载如何，对 OSD 进程执行 scrub 操作的最大时间间隔，默认每周一次
osd deep scrub interval	7*60*60*24s	对 Ceph 集群执行深度（读所有数据）scrub 操作的时间间隔，深度 scrub 不受 osd scrub load threshold 影响
osd op threads	2	为 OSD 进程设置处理服务请求的线程数目，线程越大则请求处理数读越快
osd client op priority	63	相对于 OSD 数据恢复操作的客户端操作优先级，允许值为 0~63
osd recovery op priority	10	相对于客户端操作的数据恢复操作优先级，允许值为 0~63
osd scrub priority	5	相对于客户端操作的 scrub 操作优先级，允许值为 0~63
osd snap trim priority	5	相对于客户端操作的 snap trim 操作优先级，允许值为 0~63
osd op thread timeout	30s	OSD 进程的操作线程超时时间

Ceph OSD 进程在响应客户端请求时，允许用户对处理客户端请求的线程进行操作设

置，如果 `osd op threads` 被用户设置为 0，则 Ceph OSD 进程的多线程功能被关闭，通常不建议关闭多线程，这样会降低客户端请求处理速度。默认情况下，Ceph 使用双线程处理请求，并且线程处理请求的超时时间为 30s，用户可以在数据恢复操作与客户端请求响应操作之间设置不同的优先级权重值，以确保在集群数据恢复期间得到最佳的性能。与 Ceph 线程操作相关的配置选项如表 14-5 所示。

表 14-5 Ceph OSDs 进程操作配置选项

配置选项	默认值	选项说明
<code>osd op threads</code>	2	为 OSD 进程设置处理服务请求的线程数目，线程越大则请求处理数越快
<code>osd client op priority</code>	63	相对于 OSD 数据恢复操作的客户端操作优先级，允许值为 0~63
<code>osd recovery op priority</code>	10	相对于客户端操作的数据恢复操作优先级，允许值为 0~63
<code>osd scrub priority</code>	5	相对于客户端操作的 scrub 操作优先级，允许值为 0~63
<code>osd snap trim priority</code>	5	相对于客户端操作的 snap trim 操作优先级，允许值为 0~63
<code>osd op thread timeout</code>	30s	OSD 进程的操作线程超时时间

当用户向 Ceph 存储集群添加 OSD 或删除 OSD 后，CRUSH 算法将重新平衡分布集群内部数据，在新增 OSD 时，CRUSH 算法将向新增的 OSD 进程移动 PG，在删除 OSD 时，CRUSH 算法会把欲删除 OSD 进程上的 PG 迁移至剩余 OSD 进程上。由于 PG 及其内部对象的迁移占用集群内部的资源及网络带宽，因此 PG 的迁移过程会导致 Ceph 存储集群的操作性能受到影响。为了保证集群读写操作的最佳性能，Ceph 采用“回填（Backfilling）”方式来迁移 PG，Backfilling 允许用户设置更低的 PG 迁移优先级，以便集群有更高的优先级去处理客户端的数据读写请求。与 Ceph Backfilling 相关的配置选项如表 14-6 所示。

表 14-6 Ceph OSD 进程 Backfilling 选项

配置选项	默认值	选项说明
<code>osd max backfills</code>	1	单个 OSD 进程允许的最大 backfills 数目
<code>osd backfill scan min</code>	64	每个 backfill 所能扫描的最小对象数目
<code>osd backfill scan max</code>	512	每个 backfill 所能扫描的最大对象数目
<code>osd backfill full ratio</code>	0.85	当 OSD 进程的 full ratio 值（已用容量与总容量的百分比）高于此参数值时，Ceph 拒绝执行 backfill 操作
<code>osd backfill retry interval</code>	10s	重新发起 backfill 请求的等待时间

随着 Ceph 集群的持续运行，Ceph OSD 进程的历史 Map 信息（即 OSD Map epochs）会越积越多，为了保证 OSD 进程的大量 epochs 信息不影响 Ceph 集群的性能，Ceph 提供了与 OSD Map 相关的设置选项，如表 14-7 所示。

表 14-7 Ceph OSD Map 设置选项

配置选项	默认值	选项说明
osd map dedup	True	是否允许删除 OSD Map 中的重复数据
osd map cache size	500M	存储 OSD Map 信息的缓存空间大小
osd map cache bl size	50M	OSD 守护进程中缓存 OSD Map 的内存大小
osd map cache bl inc size	100M	OSD 守护进程中缓存 OSD Map 的内存增量大小
osd map message max	100M	每个 OSD Map 消息允许的最大 Map 条目

Ceph 存储集群启动或 OSD 进程故障重启后，该 OSD 进程将与其他 OSD 进行 peering 操作，而在此期间客户端不被允许写数据。通常而言，当故障 OSD 重启恢复后，其上的数据对象和 OSD Map 信息已无法与其他 OSD 上的最新数据对象和 Map 信息同步，此时数据时间戳已落后的 OSD 进程进入数据恢复模式，并开始在其他 OSD 之间寻找数据对象的最新副本，同时更新 OSD Map 信息。如果故障 OSD 数目较大，或者 OSD 故障时间较久，则数据恢复过程将花费较长时间同时也会消耗较多的集群资源。为了保证集群在 degrade 状态时的数据操作性能，Ceph 提供了配置选项以限制发起数据恢复操作的请求数目、处理恢复操作的线程数目以及数据恢复时的数据操作块大小，如表 14-8 所示。

表 14-8 Ceph OSD 进程数据恢复配置选项

配置选项	默认值	选项说明
osd recovery delay start	0s	Peering 完成后，延迟进行数据恢复的时间
osd recovery max active	15	每个 OSD 进程一次性发起的恢复请求数目，值越大恢复速度越快，但是对集群性能影响也越大
osd recovery max chunk	8<<20	恢复时数据块大小
osd recovery threads	1	处理数据恢复的线程数
osd recovery thread timeout	30s	数据恢复线程的超时时间
osd recover clone overlap	True	数据恢复期间保留重叠的克隆数据

6. Ceph 日志设置

Ceph 存储集群使用日志机制来提升集群性能并保证数据一致性，Ceph 操日志使得 Ceph OSD 进程可以快速提交客户端的小块写入。Ceph 将小块随机 I/O 顺序写入日志，并将需要更多时间进行的合并小块写入操作留给后端文件系统，继而加速对 Ceph 存储集群的突发访问速度。此外，为了保证写入数据的一致性，Ceph OSD 进程采用了可以保证操作原子性的文件系统接口，Ceph OSD 进程将把对操作的描述写入日志，同时将操作应用到文件系统，这一机制使得数据对象（如 PG 元数据）可以实现原子性更新。每隔几秒（时间位于 filestore min sync interval 与 filestore max sync interval 之间），Ceph 周期性地将日志文件记录的操作描述应用到文件系统（即日志同步操作），继而 OSD 进程便可裁掉日志文件中已应

用的操作并可重复使用日志空间。如果同步操作失败，则 OSD 进程将在最后一次同步失败的地方重放日志。Ceph OSD 进程提供了与日志相关的配置选项，如表 14-9 所示。

表 14-9 Ceph OSD 进程日志配置选项

配置选项	默认值	选项说明
filestore max sync interval	5s	同步操作最大时间间隔
filestore min sync interval	0.01s	同步操作最小时间间隔
journal dio	True	写日志时启用直接 IO
journal aio	True	异步 IO 写入日志时启用 libaio，必须启用 dio 才能生效
journal block align	True	块对齐写操作，当使用 dio 和 aio 时，必须设为 true
journal max write bytes	10<<20	任意时刻写日志的最大字节
journal max write entries	100	任意时刻写日志的最大条目
journal queue max ops	500	队列中允许的最大操作数
journal queue max bytes	10<<20	队列中允许的最大字节数

7. Ceph Pool&PG&CRUSH 设置

当用户在 Ceph 存储集群中创建 POOL，并为 POOL 创建 PG 时，如果用户未指定具体的参数值，则 Ceph 使用配置文件中的默认值来创建 POOL 和设置 PG 数目。通常情况下，建议用户根据实际情况在配置文件中自定义 POOL 的对象副本数目（osd pool default size 与 osd pool default min size）和 PG 数目（osd pool default pg num 与 osd pool default pgp num）。关于对象副本数目，用户可以根据自身对数据安全性的要求程度进行设置，Ceph 默认存储一份主对象数据和两份副本数据（即 osd pool default size=3）。对于 PG 数目，假设数据对象副本数目为 N，集群 OSD 数目为 M，每个 OSD 上的 PG 数目为 X（官方推荐为 100），则官方推荐的 PG 数目计算公式为：

Number of PG/PGP = M*100/N

上述公式计算出的值通常不是 2 的幂次方值，为了最大优化 CRUSH 算法，通常将 PG 数目取 2 的幂次方值，如 Ceph 集群有 100 个 OSD，副本数目为 3，则实际算出的 PG 数目为 3333.3，而与此值最接近的 2 的幂次方值为 4096，因此最终的 PG 数目为 4096。POOL 的对象副本数和 PG 数目设置通常位于配置文件的 [global] 配置段，如下：

```
[global]
osd pool default size = 4
osd pool default min size = 1
osd pool default pg num = 4096
osd pool default pgp num = 4096
```

当然，用户也可以在创建 POOL 时指定对象副本数和 PG 数目，还可以在后期运维中通过 ceph osd pool_set {pool-name} size {size} 命令修改副本数目。此外，Ceph 还提供了很

多与 POOL、PG 和 CRUSH 相关的配置选项，其中比较常见的配置选项如表 14-10 所示。

表 14-10 Ceph POOL&PG&CRUSH 配置选项

配置选项	默认值	选项说明
mon max pool pg num	65536	每个 POOL 允许的最大 PG 数目
mon pg create interval	30s	在相同 OSD 进程上创建 PG 的时间间隔
osd pool default size	3	每个 POOL 中默认的对象副本数
osd pool default min size	0	POOL 中写对象副本的最小值
osd pool default pg num	8	POOL 中默认的 PG 数目
osd pool default pgp num	8	POOL 中默认的 PGP 数目
osd pool default flags	0	新建 POOL 时默认的标志
osd crush chooseleaf type	1	CRUSH 规则中用于 chooseleaf 的 bucket 类型
osd crush initial weight	OSD 对应卷的大小，单位为 TB	新加入 CRUSHMAP 中的 OSD 权重
osd pool default crush replicated ruleset	CEPH_DEFAULT_CRUSH_REPLICATED_RULESET	创建副本 POOL 时默认使用的 CRUSH 规则集

14.1.3 Ceph CRUSH 自定义

CRUSH (Controlled Replication Under Scalable Hashing) 是 Ceph 存储集群使用的一种数据分发算法，类似于 OpenStack 的 Swift 和 AWS 的对象存储所使用的哈希和一致性哈希数据分布算法。Ceph 采用 CRUSH 而非哈希算法的主要原因，在于当数据增长时哈希不能动态增加 Bucket，而一致性哈希在增加 Bucket 时需要进行大量的数据迁移。此外，其他常见的数据分发算法依赖中心化的 Metadata 服务器来存储元数据，因此造成了数据存取的效率较低，而 CRUSH 则是通过接受多维参数并通过一定的计算来解决数据动态分发的问题。与其他分布式存储集群相比，CRUSH 算法可以认为是 Ceph 存储集群的核心设计之一，Ceph 使用 CRUSH 算法对客户端对象数据进行分布存储，这也是 Ceph 存储集群区别于其他分布式存储系统并具备高扩展性、高可用性和高性能的主要原因。在本书 10.3.3 节中介绍了 Ceph 数据存储的工作原理，当 Ceph 客户端将文件存储到 Ceph 集群时，文件被条带化切割为特定大小的对象块 (Object)，对象块经过 HASH 之后得到存放对象的 PGID，之后再经过 CRUSH 算法将对象存储到 Ceph 集群的存储设备中 (即 OSD 中)，CRUSH 算法将数据 X 存储到一组 OSD 集合中的伪随机映射过程可简单表述如下：

$$CRUSH(X) \rightarrow (OSD1、OSD2、OSD3 \dots OSDn)$$

其中，n 为 Ceph 存储池中数据存储的副本数目。在 Ceph 的数据存储过程中，CRUSH 算法提供了配置可更改和数据动态再平衡等关键特性，CRUSH 算法存储数据对象的过程可通过 CRUSH Map 控制并进行自定义修改，CRUSH Map 由不同层次的逻辑 Buckets 和

Devices 组成，其中的 Devices 主要指 OSDs 存储设备，是 CRUSH Map 层次结构中的叶子节点，而层次结构中叶子节点外的全是虚拟 Bucket 设备，CRUSH Map 的层次结构如图 14-2 所示。图 14-2 中，有 Root、Datacenter、Room、Rack、Host 和 OSD 几种 Bucket 其中 Root 包含的 item 是 Datacenter (即数据中心)，而 Datacenter 包含的 item 是 Room (即机房)，Room 包含的 item 是 Rack (即机柜)，Rack 包含的 item 是 Host (即主机服务器)，最后 Host 包含的 item 才是最终的存储设备 OSD。对于每个 Ceph 集群而言，CRUSH Map 在正式上线前是已经确定的，因此如果用户需要自定义更改 CRUSH Map 的层次结构，必须在上线前进行更改并核实更改已成功应用到 CRUSH 算法中。

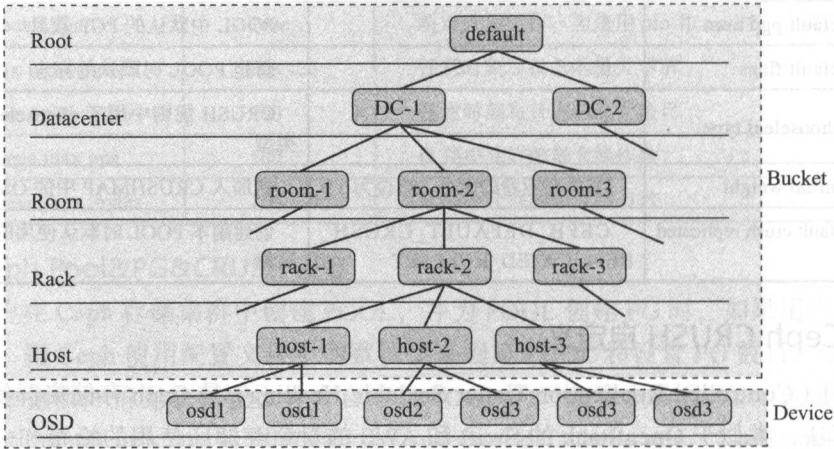


图 14-2 CRUSH Map 层次结构图

CRUSH Map 中的 Bucket 是用户可以自定义增加的，每个层级的 Bucket 对应不同的故障域，在实际应用中，为了更加精细化地隔离故障域，用户还可以增加 PDU、ROW 和 CHASSIS 等 Bucket (Bucket 的名称可以自定义)，例如在 Ceph 0.80.7 版本中，Ceph 默认的 CRUSH Map 中便声明了如下的 11 种 Bucket:

```
type 0 osd
type 1 host
type 2 chassis
type 3 rack
type 4 row
type 5 pdu
type 6 pod
type 7 room
type 8 datacenter
type 9 region
type 10 root
```

在 CRUSH 算法中，决定数据应该如何分布以及分布到哪个 Bucket 中，是通过 Placement Rule 来决定的。换言之，Ceph 存储管理员可以将用户数据分布存储到不同的数据中心、机

房、机柜和存储节点上，例如生产环境中 Ceph 默认的数据副本数目为 3，则管理员可以通过修改 CRUSH Map 中的 Placement Rule，将相同数据的三个副本分别存储到不同的故障域中，这里的故障域可以是不同的数据中心、不同的机房、不同的机柜、不同的主机甚至是不同地理位置的数据中心，从而用户可以根据数据的不同安全级别将其实现不同程度的故障域保护。在 CRUSH Map 中，每个 Rule 均定义了一系列操作，其中 Take 操作定义了入口 Bucket，即 CRUSH Map 层级结构的根节点（CRUSH Map 中可以有多个 Rule，因此可能有多个根），而 Select (N, Type) 操作定义了选取 N 个类型为 Type 的项目，最后的 Emit 操作则是提交选择的结果，CRUSH Map 中的 Rule 定义语句如下所示：

```
.....
rule<rulename> {
  ruleset<ruleset>          //rule ID
  type [ replicated | raid4 ]
  min_size <min-size>        //数据最小副本数
  max_size <max-size>        //数据最大副本数
  step take <bucket-type> //从bucket-type开始挑选存储设备
  step [choose|chooseleaf]firstn <N>type <bucket-type> //挑选规则
  step emit
}
```

其中，step take 和 step emit 之间的语句即是 Rule 定义的 Select 操作语句。由于 Bucket 包含的项目可以全部是 Bucket，也可以全部是 Device，因此如果 Select 操作中有如下语句：

```
step choose firstn <N>type <bucket-type>
```

则表示从 <bucket-type> 层级的 Bucket 中选择 N 个项目，这些项目依然是 Bucket。而如果 Select 操作中有如下语句：

```
step chooseleaf firstn <N>type <bucket-type>
```

则表示从 <bucket-type> 层级的 Bucket（通常为 Host）中选择 N 个叶子节点，即 OSD 层级设备。因此，在 Rule 的 Select 操作中，不管有多少个 choose 操作，最后一定以 chooseleaf 选择 OSD 存储设备结尾。此处的 <N> 值可能有三种情况：

- ❑ 等于 0：从 <bucket-type> 层级的 Bucket 中选择与 POOL 副本数目相同的项目，例如 Ceph 存储池默认副本数目为 3，如果 N 为 0，并且是 chooseleaf 操作，而且 type 为 host，则表明从不同的 host 中选择 3 个 OSD 用来存储对象数据及其副本。
- ❑ 大于 0：从 <bucket-type> 层级的 Bucket 中选择 N 个项目，例如 Ceph 存储池默认副本数目为 3，如果 N 为 1，并且是 chooseleaf 操作，而且 type 为 host，则表明从 host 层级的 Bucket 中选择 1 个 OSD 用来存储对象数据，注意此时另外两个副本数据需要从其他的 Bucket 中选择 OSD 来存放。
- ❑ 小于 0：小于 0 的情况通常与大于 0 操作同时使用，即同一个 Rule 中有 N>0 的 Step，再有 N<0 的 Step。假设存储池的副本数目为 M，而 N<0 的 Step 通常表示从 <bucket-type> 层级的 Bucket 中选择 M-|N| 个项目。例如 Ceph 存储池默认副本数

目为 3，如果 N 为 -1，并且是 chooseleaf 操作，而且 type 为 host，则表明从 host 层级的 Bucket 中选择 $3 - |-1| = 2$ 个 OSD 用来存储对象数据，注意此时另外一个 OSD（通常为 Primary OSD）可能已经通过 $N > 1$ 的 Step 进行了选取。

CRUSH Map 中可以有多个 Rule，而每个 Rule 都对应有自己的 ID。此外，Ceph 客户端通过 POOL 来访问 Ceph 存储集群，而每个 POOL 都有对应的 Rule ID，当客户端数据写入此 POOL 时，Ceph 将从 CRUSH Map 中获取与此 POOL 对应的 Rule，并根据 Rule 定义的操作将数据分布存储到不同的 Bucket 和 Device 中。为了便于讲解，本节继续使用第 10 章部署的 Ceph 存储集群进行分析演示，在该集群中，OpenStack 计算节点 compute1 和 compute2，网络节点 network1 和 network2 以及存储节点 storage1 均作为 Ceph 存储节点，该 Ceph 实验集群共有 5 个 OSD、576 个 PG 和 6 个 POOL，Ceph 存储集群的运行情况如下所示：

```
[root@controller1 ~]# ceph -s
cluster 4042b616-be34-44c1-b806-8a80f1fcd11d
health HEALTH_OK
monmap e1: 3 mons at {compute1=192.168.142.44:6789/0,compute2=192.168.142.45:6789/0,
storage1=192.168.142.46:6789/0}, election epoch 102, quorum 0,1,2 compute1,
compute2,storage1
osdmap e81: 5 osds: 5 up, 5 in
pgmap v515: 576 pgs, 6 pools, 147 MB data, 46 objects
643 MB used, 45381 MB / 46025 MB avail
576 active+clean

[root@controller1 ~]# ceph osd lspools
0 data,1 metadata,2 rbd,3 volumes,4 vms,5 images,
```

为了查看 Ceph 集群的 CRUSH Map 信息，首先需要通过 getcrushmap 命令获取集群的 CRUSH Map，如下：

```
[root@controller1 ~]# ceph osd getcrushmap -o compiled-crushmap.txt
got crush map from osdmap epoch 81
```

getcrushmap 命令获取到的 CRUSH Map 是编译过的二进制信息，并不能直接编辑或查看，必须使用 crushtool 命令行工具对其进行反编译，如下：

```
[root@controller1 ~]# crushtool -d compiled-crushmap.txt -o \
decompiled-crushmap.txt
[root@controller1 ~]# ls -l decompiled-crushmap.txt
-rw-r--r-- 1 root root 1450 Feb  1 23:10 decompiled-crushmap.txt
```

经过反编译的 CRUSH Map 即可进行编辑修改，当前 Ceph 集群（版本为 0.80.7）默认的 CRUSH Map 信息如下：

```
[root@controller1 ~]# more decompiled-crushmap.txt
# begin crush map
tunable choose_local_tries 0
tunable choose_local_fallback_tries 0
tunable choose_total_tries 50
```

```

tunable chooseleaf_descend_once 1
//Ceph存储集群中的Devices, 即OSDs
# devices
device 0 osd.0
device 1 osd.1
device 2 osd.2
device 3 osd.3
device 4 osd.4

// Buckets名称声明, 一共11个
# types
type 0 osd
type 1 host
type 2 chassis
type 3 rack
type 4 row
type 5 pdu
type 6 pod
type 7 room
type 8 datacenter
type 9 region
type 10 root

//具体的Bucket实例定义
# buckets
host compute1 {
id -2 # do not change unnecessarily
# weight 0.010
alg straw
hash 0 # rjenkins1
item osd.0 weight 0.010 //Bucket包含的项目
}
host compute2 {
id -3 # do not change unnecessarily
# weight 0.010
alg straw
hash 0 # rjenkins1
item osd.1 weight 0.010 //Bucket包含的项目
}
host network1 {
id -4 # do not change unnecessarily
# weight 0.010
alg straw
hash 0 # rjenkins1
item osd.2 weight 0.010 //Bucket包含的项目
}
host network2 {
id -5 # do not change unnecessarily
# weight 0.010
alg straw
hash 0 # rjenkins1

```



```

        item osd.3 weight 0.010 //Bucket包含的项目
    }
    host storage1 {
    id -6 # do not change unnecessarily
        # weight 0.010
    alg straw
    hash 0 # rjenkins1
        item osd.4 weight 0.010 //Bucket包含的项目
    }
    root default {
    id -1 # do not change unnecessarily
        # weight 0.050
    alg straw
    hash 0 # rjenkins1
        item compute1 weight 0.010 //Bucket包含的项目
        item compute2 weight 0.010 //Bucket包含的项目
        item network1 weight 0.010 //Bucket包含的项目
        item network2 weight 0.010 //Bucket包含的项目
        item storage1 weight 0.010 //Bucket包含的项目
    }

//placement rule定义
# rules
rule replicated_ruleset {
ruleset 0 //Rule ID
type replicated
    min_size 1 //最小副本数, 如果小于此值, 则该规则无效
    max_size 10 //最大副本数, 如果大于此值, 则该规则无效
    step take default //从default这个bucket开始遍历挑选存储设备
    step chooseleaf firstn 0 type host //从全部host中选择3个存储设备
step emit
}
# end crush map

```

从上述 CRUSH Map 中可以看到, Ceph 默认使用的故障域为 Host, 即 Ceph 默认将存储池中的三份数据副本分别存储在三个不同主机上, CRUSH 规则中的 Select (0,host) 操作 (即 step chooseleaf firstn 0 type host) 表示从 Host 故障域 (即 Host Bucket) 中选择三个存储设备 (Host Bucket 包含的项目是 Devices, 而 Ceph 存储池默认的 pool_size 为 3)。为了对其验证, 可以查看 ID 为 0 的 POOL (即默认创建的数据存储池) 中 PG 与 OSDs 之间的映射关系, 结果如下:

```

[root@controller1 ~]# ceph pg dump|grep ^0\.|awk '{print $1 "\t" $16}'
dumped all in format plain
.....
0.14    [4,3,2]
0.13    [3,4,0]
0.12    [3,4,2]
0.11    [1,3,0]
0.10    [4,0,2]

```

0.f	[0,3,1]
0.e	[0,4,2]
0.d	[2,3,0]
0.c	[1,4,0]
0.b	[0,1,4]
0.a	[2,3,4]
0.9	[3,1,0]
0.8	[1,2,4]
0.7	[3,2,0]
0.6	[0,2,1]
0.5	[3,4,0]
0.4	[1,2,0]
0.3	[1,0,2]
0.2	[1,4,2]
0.1	[1,3,4]
0.0	[4,2,3]

其中，“0.14”中的“0”表示存储池 ID，“14”表示 PG ID，而“[4,3,2]”表示 ID 为 0 的存储池中 ID 为 14 的 PG 被以三副本形式存储到 OSD.4、OSD.3 和 OSD.2 中（其中 OSD.4 是 Primary OSD），由于这三个 OSD 分别位于不同的主机上，因此任一主机故障都不会影响 PG 的存储，即实现了主机故障域的隔离。

为了实现更高层级的故障域，如 Rack 级别的故障域，用户可以自定义 CRUSH Map。现假设存储节点 compute1 和 compute2 位于 Rack-1 上，network1 和 network2 位于 Rack-2 上，而 storage1 位于 Rack-3 上，同时每个存储节点上有一个 OSD 进程，Ceph 存储集群的硬件环境配置如表 14-11 所示。

表 14-11 Ceph 存储集群硬件配置

存储节点名称	Rack 编号	OSD 编号
compute1	Rack-1	OSD.0
compute2	Rack-1	OSD.1
network1	Rack-2	OSD.2
network1	Rack-2	OSD.3
storage1	Rack-3	OSD.4

现在，为 Ceph 集群自定义 CRUSH Map 实现 Rack 级别的故障域，要求将存储池中数据对象的三个副本分别存储到位于 Rack-1、Rack-2 和 Rack-3 上的存储节点中。由于 Ceph 默认使用的是 Host 层级的 Bucket，根据表 14-11 中的配置，需要在默认的 CRUSH Map 中添加三个 Rack 层级的 Bucket，同时三个新增的 Bucket 包含的项目分别为 compute1 和 compute2、network1 和 network2 以及 storage1，如下：

```
rack rack-1{
  id -7 # do not change unnecessarily
  # weight 0.010
```

```

alg straw
hash 0 # rjenkins1
item compute1 weight 0.010
item compute2 weight 0.010
}
rack rack-2{
id -8 # do not change unnecessarily
# weight 0.010
alg straw
hash 0 # rjenkins1
item network1 weight 0.010
item network2 weight 0.010
}
rack rack-3{
id -9 # do not change unnecessarily
# weight 0.010
alg straw
hash 0 # rjenkins1
item storage1 weight 0.010
}

```

修改默认的 Root 层级，默认 Root 层级名称为 default，其包含的项目为 Host 层级的 Bucket，将其修改为 Rack 层级的 Bucket，如下：

```

root default {
id -1 # do not change unnecessarily
# weight 0.050
alg straw
hash 0 # rjenkins1
item rack-1 weight 0.020
item rack-2 weight 0.020
item rack-3 weight 0.010
}

```

修改默认的 Placement Rule，将默认的 Host 层级故障域修改为 Rack 层级故障域，如下：

```

rule replicated_ruleset {
ruleset 0
type replicated
min_size 1
max_size 10
step take default
step chooseleaf firstn 0 type rack //将默认的host修改为rack
step emit
}

```

至此，基于 Rack 层级故障域的 CRUSH Map 已自定义完成，完整的 CRUSH Map 信息如下：

```

[root@controller1 ~]# moredecompiled-crushmap.txt
# begin crush map

```

```

tunable choose_local_tries 0
tunable choose_local_fallback_tries 0
tunable choose_total_tries 50
tunable chooseleaf_descend_once 1

# devices
device 0 osd.0
device 1 osd.1
device 2 osd.2
device 3 osd.3
device 4 osd.4

# types
type 0 osd
type 1 host
type 2 chassis
.....

# buckets
host compute1 {
id -2    # do not change unnecessarily
        # weight 0.010
    alg straw
    hash 0  # rjenkins1
    item osd.0 weight 0.010
}
host compute2 {
id -3    # do not change unnecessarily
        # weight 0.010
    alg straw
    hash 0  # rjenkins1
    item osd.1 weight 0.010
}
host network1 {
id -4    # do not change unnecessarily
        # weight 0.010
    alg straw
    hash 0  # rjenkins1
    item osd.2 weight 0.010
}
host network2 {
id -5    # do not change unnecessarily
        # weight 0.010
    alg straw
    hash 0  # rjenkins1
    item osd.3 weight 0.010
}
host storage1 {
id -6    # do not change unnecessarily
        # weight 0.010
    alg straw

```

```

hash 0 # rjenkins1
item osd.4 weight 0.010
}
rack rack-1{
id -7      # do not change unnecessarily
           # weight 0.010

alg straw
hash 0 # rjenkins1
item compute1 weight 0.010
item compute2 weight 0.010
}
rack rack-2{
id -8      # do not change unnecessarily
           # weight 0.010

alg straw
hash 0 # rjenkins1
item network1 weight 0.010
item network2 weight 0.010
}
rack rack-3{
id -9      # do not change unnecessarily
           # weight 0.010

alg straw
hash 0 # rjenkins1
item storage1 weight 0.010
}
root default {
id -1      # do not change unnecessarily
           # weight 0.050

alg straw
hash 0 # rjenkins1
item rack-1 weight 0.020
item rack-2 weight 0.020
item rack-3 weight 0.010
}
# rules
rule replicated_ruleset {
ruleset 0
type replicated
    min_size 1
    max_size 10
step take default
step chooseleaf firstn 0 type rack
step emit
}
# end crush map

```

将新的 CRUSH Map 应用到 Ceph 集群前，需要将其编译为二进制文件，如下：

```
[root@controller1 ~]# crushtool -c decompiled-crushmap.txt -o crushmap_new
```

通过 `setcrushmap` 命令将用户自定义的 CRUSH Map 应用到 Ceph 集群，如下：

```
[root@controller1 ~]# ceph osd setcrushmap -i crushmap_new
set crush map
```

应用新的 CRUSH Map 后，Ceph 集群中的 PG 开始重新分布，集群内部的带宽和 IO 都会迅速增加。PG 重新分布一定时间后（取决于集群网络带宽和磁盘 IO 性能），所有 PG 恢复到 active+clean 状态。此时，可以查看集群中的 OSD 映射树以验证 CRUSHMap 设置是否生效，如下：

```
[root@storage1 ~]# ceph osd tree
# id      weight type name      up/down reweight
-1        0.04997 root default
-7        0.01999      rack rack-1
-2        0.009995             host compute1
  0        0.009995             osd.0   up      1
-3        0.009995             host compute2
  1        0.009995             osd.1   up      1
-8        0.01999      rack rack-2
-4        0.009995             host network1
  2        0.009995             osd.2   up      1
-5        0.009995             host network2
  3        0.009995             osd.3   up      1
-9        0.009995      rack rack-3
-6        0.009995             host storage1
  4        0.009995             osd.4   up      1
```

默认 CRUSHMap 以 Host 层级的 Bucket 为故障域，因此 PG 在 OSD.0~OSD.4 五个 OSD 存储设备中随机选取三个 OSD 存储 PG，由于新的 CRUSH Map 采用了 Rack 层级故障域，而 Rack-3 中的存储节点 Storage1 仅有 OSD.4 一个存储设备，因此应用新的 CRUSHMap 后，对于三副本存储池中的任何一个 PG，必须至少有一份数据存储到 OSD.4 中。表 14-12 反应了这种变化，应用新的 CRUSHMap 前，ID 为 0 的 POOL 中有部分 PG 未映射到存储设备 OSD.4 中，但是应用新的 CRUSHMap 后，该存储池中全部 PG 都有且仅有一份数据映射到了 OSD.4 中。

表 14-12 新 CRUSHMap 应用前后 PG 与 OSD 映射对比

应用新的 CRUHS Map 前		应用新的 CRUHS Map 后	
PG	OSD	PG	OSD
0.3	[1,0,2]	0.3	[1,4,2]
0.4	[1,2,0]	0.4	[1,3,4]
0.6	[0,2,1]	0.6	[4,2,0]
0.7	[3,2,0]	0.7	[4,0,2]
0.9	[3,1,0]	0.9	[3,1,4]
0.d	[2,3,0]	0.d	[4,1,3]
0.f	[0,3,1]	0.f	[0,2,4]
0.11	[1,3,0]	0.11	[1,3,4]

14.1.4 Ceph SSD 应用场景

随着存储技术的不断发展，SSD 固态硬盘已不再是“凤毛麟角”的奢侈品，很多中小企业和个人均能享受到 SSD 带来的读写高性能。在 Ceph 存储集群中，合理利用 SSD 硬盘可以为 Ceph 集群带来极大的性能改善，通过 SSD 与 SAS 或 SATA 等机械硬盘的混用，Ceph 可以将不同应用场景的数据区别存储到不同性能的硬盘上，如 IO 密集型应用存储到 SSD 硬盘上，而备份数据则存储到机械硬盘上。除此之外，SSD 固态硬盘在 Ceph 存储集群中还有其他使用方式，如作为 OSD 的日志盘、作为 Primary OSD 以及作为 Ceph 的 Cache Tier 层等，下面对 SSD 的这几种使用方式进行简要介绍。

(1) SSD 作为 OSD 日志盘

使用 SSD 作为 OSD 的日志盘是 Ceph 官方推荐的使用方式，Ceph 使用日志来保证数据的一致性并提高存储性能，利用高速 SSD 固态硬盘存储 Ceph OSD 日志是最为常见的 Ceph 存储集群配置与部署方式。Ceph 集群中的 OSD 进程可以将日志写入单块 SSD 硬盘或 SSD 硬盘的分区中，在多个 OSD 同时写单块 SSD 固态硬盘的分区时，影响 Ceph 性能的关键因素主要是 SSD 日志盘的顺序读写性能，因此在选择 SSD 硬盘时务必对其进行顺序读写性能的测试，同时在对 SSD 硬盘分区时，最好对其进行分区对齐，这样可获取最佳的性能。此外，为了安全起见，作为日志盘的 SSD 硬盘应该禁用写缓存功能（hdparm -W 0 /dev/hda 0）。SSD 硬盘作为 Ceph OSD 日志盘的示例如图 14-3 所示。

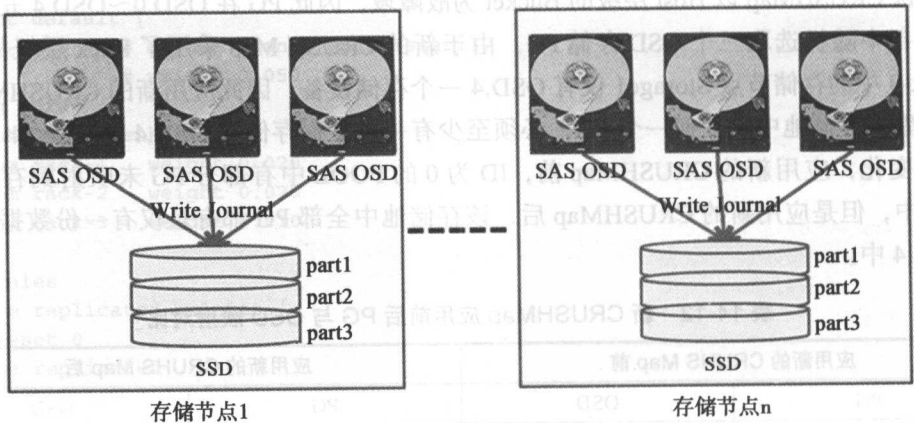


图 14-3 SSD 固态硬盘分区作为 OSD 日志

(2) SSD 与 SAS/SATA 混用组成快慢存储池

为了满足不同的应用场景需求，可以在同一个 Ceph 存储集群中配置不同性能的存储池，如将 SSD 固态硬盘组成高性能 POOL，而将 SAS/SATA 盘组成常规 POOL。在 OpenStack 环境中，Nova 实例对实时数据 IO 要求较高，而且都是处于活跃状态的热数据，可以将 SSD 组成的 Ceph 存储池作为 Nova 的存储后端用于存储虚拟机实例，而将活跃程度不高的冷数据，如 Glance 镜像数据和 Cinder 块设备备份数据存储到 SAS/SATA 盘组成的常

规 Ceph 存储池中。SSD 与 SAS/SATA 混合使用组成快慢 Ceph 存储池的示例如图 14-4 所示，其中，每个 Ceph 存储节点上既配置了 SSD 硬盘，也配置了 SAS 硬盘，而全部存储节点上的 SSD 硬盘共同组成 SSD 存储池，同时全部存储节点上的 SAS 硬盘共同组成了 SAS 存储池，这样便可将 Nova 虚拟机实例存储到 SSD 存储池中，而将镜像和备份数据存储到 SAS 存储池中。

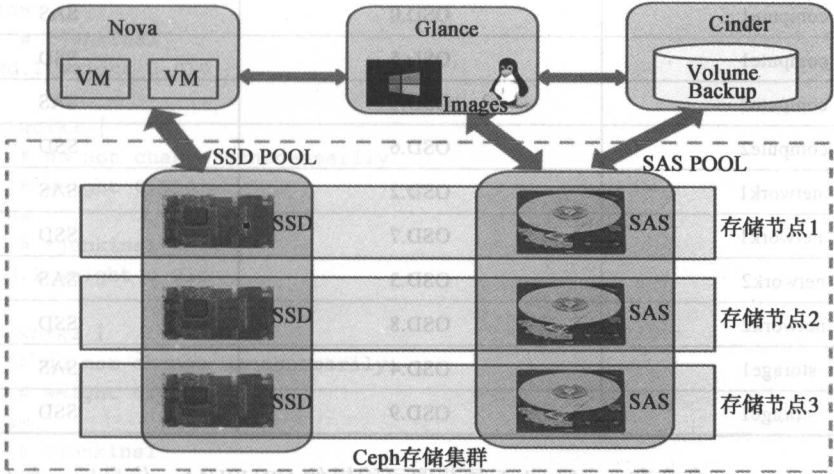


图 14-4 SSD 与 SAS 混用组成快慢存储池

SSD 与 SAS/SATA 混合使用组成不同性能的存储池，主要通过修改 Ceph 的 CRUSHMap 来实现。默认情况下，Ceph 仅有一个 Root 层级，在该 Root 层级下，所有 Device 被看成是同一类型的磁盘，而为了区分 SSD 和 SAS 硬盘，需要在 CRUSHMap 中新增一个 Root 层级。为了修改方便，可以将原 Root 层级 Bucket 的名称由 default 修改为 sas，并将存储节点中 SAS 类型的 Device 归入 sas 层级中，此外再新增一个 Root 层级的 Bucket 取名为 ssd，同时将存储节点中 SSD 类型的 Device 归入 ssd 层级中，如图 14-5 所示。

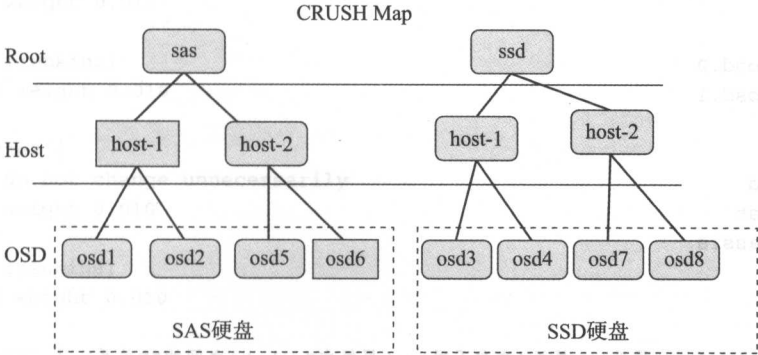


图 14-5 SSD 与 SAS 混用组成不同性能存储池时的 CRUSH Map 层级

为了讲解如何配置 SSD 与 SAS 混用组成快慢存储池的 CRUSHMap，本节仍然以第 10 章部署的 Ceph 存储集群为例，并为每个存储节点新增一个 SSD 硬盘，并将其配置为 OSD 存储设备，配置后的 Ceph 存储集群硬件配置如表 14-13 所示。

表 14-13 增加 SSD 磁盘后 Ceph 存储集群硬件配置

存储节点名称	OSD 编号	硬 盘 类 型
compute1	OSD.0	SAS
compute1	OSD.5	SSD
compute2	OSD.1	SAS
compute2	OSD.6	SSD
network1	OSD.2	SAS
network1	OSD.7	SSD
network2	OSD.3	SAS
network2	OSD.8	SSD
storage1	OSD.4	SAS
storage1	OSD.9	SSD

通过 `getcrushmap` 命令和 `crushtool` 工具获取可编辑的 CRUSHMap 信息后，在 CRUSHMap 中新建一个 Root 层级，并取名为 `ssd`，将 SSD 类型的 OSD 移入 Host 层级，同时将原 Root 层级由 `default` 修改为 `sas`，将 Root 层级的入口 Bucket 由 `default` 改为 `sas`。编辑完成后，最终的 CRUSH Map 如下：

```
[root@controller1 ~]# morecrushmap.txt
# begin crush map
tunable choose_local_tries 0
tunable choose_local_fallback_tries 0
tunable choose_total_tries 50
tunable chooseleaf_descend_once 1

# devices
device 0 osd.0
device 1 osd.1
.....
# types
type 0 osd
type 1 host
type 2 chassis
.....

# buckets
//以下host buckets属于sas层级(注意不同root层级内Buckets之间的ID关系)
host compute1 {
id -2  # do not change unnecessarily
```

```

    # weight 0.010
    alg straw
    hash 0 # rjenkins1
    item osd.0 weight 0.010
  }
  host compute2 {
    id -3 # do not change unnecessarily
    # weight 0.010
    alg straw
    hash 0 # rjenkins1
    item osd.1 weight 0.010
  }
  host network1 {
    id -4 # do not change unnecessarily
    # weight 0.010
    alg straw
    hash 0 # rjenkins1
    item osd.2 weight 0.010
  }
  host network2 {
    id -5 # do not change unnecessarily
    # weight 0.010
    alg straw
    hash 0 # rjenkins1
    item osd.3 weight 0.010
  }
  host storaget1 {
    id -6 # do not change unnecessarily
    # weight 0.010
    alg straw
    hash 0 # rjenkins1
    item osd.4 weight 0.010
  }
  //以下host buckets属于ssd层级
  host compute1 {
    id -8 # do not change unnecessarily
    # weight 0.010
    alg straw
    hash 0 # rjenkins1
    item osd.5 weight 0.010
  }
  host compute2 {
    id -9 # do not change unnecessarily
    # weight 0.010
    alg straw
    hash 0 # rjenkins1
    item osd.6 weight 0.010
  }
  host network1 {
    id -10 # do not change unnecessarily

```

```

    # weight 0.010
alg straw
hash 0 # rjenkins1
item osd.7 weight 0.010
}
host network2 {
id -11 # do not change unnecessarily
    # weight 0.010
alg straw
hash 0 # rjenkins1
item osd.8 weight 0.010
}
host storage1 {
id -12 # do not change unnecessarily
    # weight 0.010
alg straw
hash 0 # rjenkins1
item osd.9 weight 0.010
}
//SAS POOL的root bucket
root sas {
id -1 # do not change unnecessarily
    # weight 0.050
alg straw
hash 0 # rjenkins1
item compute1 weight 0.010
item compute2 weight 0.010
item network1 weight 0.010
item network2 weight 0.010
item storage1 weight 0.010
}
//SSD POOL的root bucket
root ssd {
id -7 # do not change unnecessarily
    # weight 0.050
alg straw
hash 0 # rjenkins1
item compute1 weight 0.010
item compute2 weight 0.010
item network1 weight 0.010
item network2 weight 0.010
item storage1 weight 0.010
}

# rules
//sas层级的Rule
rule sas {
    ruleset 0 //Rule ID为0
type replicated
min_size 1

```

```

max_size 10
step take sas //入口Bucket为sas
step chooseleaf firstn 0 type host
step emit
}
//ssd层级的Rule
rule ssd {
    ruleset 1 //Rule ID为1
    type replicated
    min_size 1
    max_size 10
    step take ssd //入口Bucket为ssd
    step chooseleaf firstn 0 type host
    step emit
}
# end crush map

```

通过 `crushtool` 和 `setcrushmap` 命令对编辑后的 CRUSHMap 文本文件进行二进制编译并应用到 Ceph 集群后，即可在 Ceph 存储集群中创建 SAS 和 SSD 存储池，如下：

```

[root@controller1 ~]#ceph osd pool create SSD 128 128
pool 'SSD' created
[root@controller1 ~]#ceph osd pool create SAS 128 128
pool 'SAS' created

```

存储池创建完成后，为其指定对应的 Rule（SAS 存储池对应的 Rule ID 为 0，SSD 对应的 Rule ID 为 1），如下：

```

[root@controller1 ~]#ceph osd pool set SSD crush_ruleset 1
[root@controller1 ~]#ceph osd pool set SAS crush_ruleset 0

```

至此，SSD 和 SAS 存储池已设置完成，现在用户即可将 SSD 存储池分配给热数据应用场景使用，如 OpenStack 中的 Nova 虚拟机，同时将 SAS 存储池分配给冷数据应用场景使用，如 OpenStack 中的镜像、备份和快照等。

（3）SSD 用于存储 Primary OSD 数据

SSD 另外一种应用场景是根据 Ceph 的数据存取流程，如图 14-6 所示，将 OSD 集合中的主数据存储在 SSD 上，而将其他副本数据存储在 SAS 硬盘上，这样既能提升 Ceph 的读写性能又不因为存储副本数据而浪费昂贵的 SSD 硬盘，是一种性价比较高的实现方案。

此类 SSD 应用场景与上一种类似，都是 SSD 与 SAS 硬盘混合使用，不同之处在于不用创建独立的 SSD 存储池，而只需配置 CRUSH 读写规则，使得所有数据的主副本存储在 SSD 类型的 OSD 中，而其他副本数据存储在 SAS/SATA 类型的 OSD 中，如图 14-7 所示。

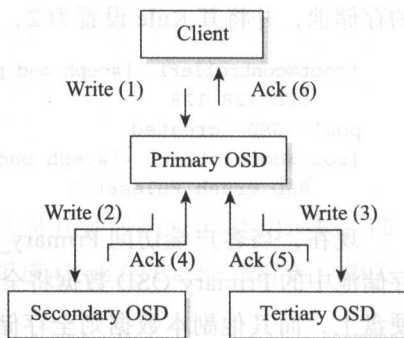


图 14-6 Ceph 客户端数据读写流程

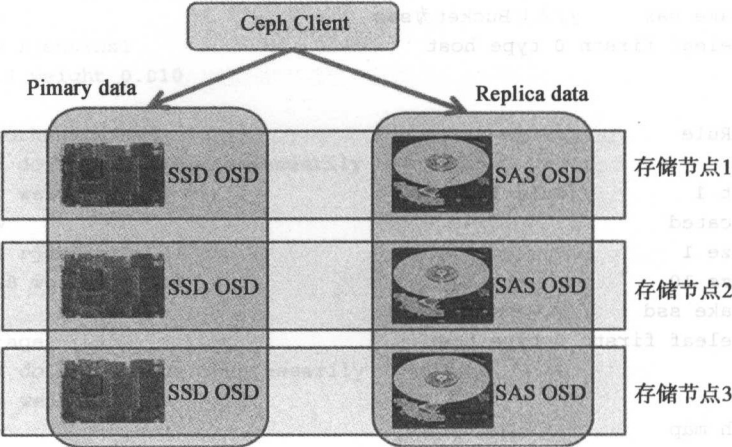


图 14-7 Ceph 主备数据分别存储到 SSD 和 SAS 硬盘 OSD 中

SSD 用于存储 Primary OSD 数据场景的关键点是 CRUSH 中 Placement Rule 的配置，此处继续沿用上一场景配置的 CRUSHMap 信息，并在其中新增如下 Rule：

```
rule primary-ssd {
    ruleset 2      //Rule ID为2
    type replicated
    min_size 1
    max_size 10
    step take ssd   //入口Bucket为ssd
    step chooseleaf firstn 1 type host //从SSD层级中选择一个OSD存储主副本数据
    step emit
    step take sas   //入口Bucket为sas
    step chooseleaf firstn -1 type host //从SAS层级中选择OSD存储剩余的副本数据
    step emit
}
```

将编辑后的 CRUSHMap 应用到 Ceph 集群后，在 Ceph 中创建一个名为 Primary_SSD 的存储池，并将其 Rule 设置为 2，如下：

```
[root@controller1 ~]#ceph osd pool create Primary_
SSD 128 128
pool 'SSD' created
[root@controller1 ~]#ceph osd pool set Primary_
SSD crush_ruleset 2
```

现在，当客户端访问 Primary_SSD 存储池时，该存储池中的 Primary OSD 数据将全部存储在 SSD 类型硬盘上，而其他副本数据则全存储在 SAS/SATA 类型硬盘上，如图 14-8 所示。

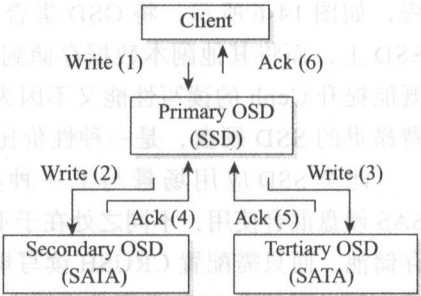


图 14-8 客户端主数据写入 SSD 中

此外，本节所介绍的主备数据分开存储方式也可通过 OSD 的亲和性来设置（0.80 后版本才支持），OSD 的亲和性可通过 `primary affinity` 参数设置，`primary affinity` 的取值范围在 0~1 之间，“0”意味着该 OSD 不会成为 Primary OSD，“1”意味着该 OSD 被指定为 Primary OSD。鉴于此，可以为 SSD 类型的 OSD 设置 `primary affinity=1`，而 SAS/SATA 类型的 OSD 设置 `primary affinity=0`，这样客户端的主数据便自动写入 SSD 中，而备数据则写入 SAS/SATA 中。默认情况下 OSD 的 Primary Affinity 功能被关闭，可通过如下方式查看：

```
[root@compute2 mon]# ceph --admin-daemon /var/run/ceph/ceph-mon.*.asok config
show |grep primary_affinity
"mon_osd_allow_primary_affinity": "false",
```

动态修改 `mon_osd_allow_primary_affinity` 参数值为 `True`，如下：

```
[root@controller1 ~]# ceph tell mon.* injectargs \ "--mon_osd_allow_primary_
affinity=1"
[root@compute1 ~]# ceph --admin-daemon /var/run/ceph/ceph-mon.*.asok config show
|grep \
primary_affinity
"mon_osd_allow_primary_affinity": "true",
```

在 `ceph.conf` 配置文件的 `[mon]` 配置段添加 `monosdallowprimaryaffinity` 配置选项，如下：

```
[mon]
monosdallowprimaryaffinity = true
```

把 SATA 类型 OSD 的 `primaryaffinity` 设置为 0，如下：

```
[root@controller1 ~]# for i in 0 1 2;do ceph osd primary-affinity osd.$i 0;done
set osd.0 primary-affinity to 0 (802)
set osd.1 primary-affinity to 0 (802)
set osd.2 primary-affinity to 0 (802)
```

SATA 类型 OSD 的亲和性被设置为 0 后，其将不会再作为 Primary OSD，可通过如下方式验证：

```
[root@controller1 ~]#for i in 0 1 2;do ceph pg dump|grep active+clean|egrep "\
[$i,]"|wc -l;done
dumped all in format plain
0
dumped all in format plain
0
dumped all in format plain
0
```

最后，还有一种目前虽然使用不是很普遍，但是有很大使用前景的 SSD 使用场景，即将 SSD 硬盘应用到 Ceph Cache Tier 技术中，用 SSD 硬盘组成 Ceph 的 Cache 缓存池，相关的配置和技术实现可参考 Ceph 官方网站的介绍^①。

① <http://docs.ceph.com/docs/master/rados/operations/cache-tiering/>

14.1.5 Ceph 性能调优关键

Ceph 就如一匹千里马，只有最好的伯乐才能发挥其最佳性能。根据相关资料的统计，Ceph 有上千个配置项可供用户设置，而为每个配置项设置不同的值，则可能影响到 Ceph 存储集群的性能，因此 Ceph 在为用户提供灵活配置的同时，也对用户提出了更高的要求。在实际部署应用中，虽然 Ceph 也提供了全部参数的默认值，但是以默认值运行的 Ceph 集群不可能获得最佳性能，因此在 Ceph 存储集群正式上线前和运行过程中，都需要根据实际运行情况对 Ceph 进行性能优化。通常而言，对 Ceph 的性能优化主要分为硬件层面和软件层面的优化，硬件层面主要在系统上线运行前进行规划配置，而软件层面的配置优化又可以分为操作系统层面和 Ceph 配置参数的优化，操作系统相关的优化最好在上线前完成，而 Ceph 配置参数可以在运行过程中动态调整。

1. 硬件配置优化

最好的硬件才能跑出最佳的性能，这点毋庸置疑，但是对于大多数客户而言，性价比才是最切实际的追求。下面对运行 Ceph 集群的硬件配置要求给出参考，用户可以根据自身情况进行适当调整。

(1) CPU

Ceph 消耗 CPU 资源最多的进程主要是 `ceph-osd` 和 `ceph-msd` 进程，`ceph-mon` 进程不会消耗太多 CPU 资源，因此生产环境下不建议 `ceph-osd` 和 `ceph-msd` 与其他应用程序共享服务器 CPU 资源。通常，为存储节点上的每个 `ceph-osd` 绑定一个 CPU 核，如果需要使用 CephFS 文件系统，则需要为 `ceph-mds` 预留尽可能多的 CPU 资源，而 `ceph-mon` 则无须预留太多 CPU 资源。

(2) 内存

生产环境中每个 `ceph-osd` 至少需要 1GB 内存，`ceph-mon` 和 `ceph-mds` 需要 2GB 内存。内存不足并不意味着 Ceph 进程不能运行，但是性能会受到很大影响。

(3) 网络

从性能角度考虑，Ceph 需要两个网络，即集群网络和客户端网络。网络带宽方面，1GB 以太网是最起码的要求，10GB 以太网是相对不错的选择，25GB 以太网相对而言是性价比最高的。然而，如果有条件，可以使用 Mellanox 生产的低延时 infiniband 网络，现存市面大多数网络设备使用高通的芯片，而 Mellanox 自研芯片其延迟是业界最低的。

(4) SSD

SSD 固态硬盘可以用作 OSD 的日志盘，也可以当作 OSD 使用。SSD 对 Ceph 集群的性能影响从两方面体现，一是 SSD 固态硬盘的选择，二是 SSD 在 Ceph 集群中的配置使用。通常从成本角度考虑，选用相对便宜的 SATA SSD 作为 OSD，而使用较贵性能也更好的 PCIe SSD 作为日志盘，如果预算允许，也可以采用更为先进的 Intel NVMe SSD 作为日志盘。

(5) 开启超线程

目前的 CPU 基本上都支持超线程 (Hyper-Threading, HT) 技术, HT 允许 CPU 并行执行操作, 在负载繁忙时最大化地利用 CPU 资源。通常, 在云计算生产环境中的服务器的 HT 和 VT 均是开启的, 假设一颗物理 CPU 有 8 核, 同时开启了 HT (双线程), 则操作系统中将看到 16 核逻辑 CPU。在 BIOS 中开启 HT 如图 14-9 所示。

(6) 关闭 NUMA

非一致性存储访问结构 (No-Uniform Memory Access, NUMA) 是目前最为常见的三大处理器结构之一^①, NUMA 的主要特点就是不存在

SMP 结构的扩展问题。NUMA 将系统 CPU、内存和 IO 插槽等划分为各自独立的区域, 各个区域彼此之间可以互联访问, 在 NUMA 结构下, CPU 访问区域内部的资源 (如内存) 要比访问其他区域内部的资源快得多, 而不像 SMP 具有共享的内存且各个 CPU 访问内存的速率完全一致, 因此 NUMA 称为非一致性访问, 而 SMP 称为一致性访问。在 Ceph 集群中, NUMA 可能会影响 ceph-osd 进程^②, 因此建议在部署 Ceph 时关闭 NUMA, NUMA 可以在 BIOS 或系统层面将其关闭, 如果是 Linux 系统, 则可以在 /etc/grub.conf 的 kernel 行最后添加 “numa=off” 将其关闭, BIOS 中关闭 NUMA 如图 14-10 所示。

(7) 关闭节能模式

很多服务器在出厂时处于节能考虑, 节能模式会被默认开启, 在节能模式下 CPU 将自动根据工作负载动态调整工作频率。不过, 根据很多运维人员的经验, 节能模式并非如想象中一样完美, 尤其是在运行高性能应用程序时, 节能模式很可能会是一个“坑”, 因此建议在部署 Ceph 的服务器上将节能模式从 BIOS 中关闭, 如图 14-11 所示。

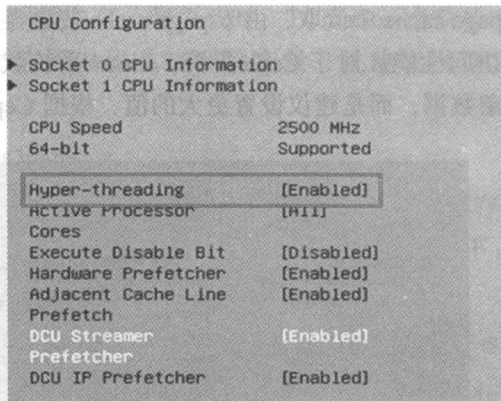


图 14-9 在 BIOS 中开启 HT

2. 操作系统优化

(1) 进程数量

Linux 内核默认的进程数量并不适合存在大量 OSD 进程的存储节点, 因此建议设置更大的 PID 值, 如下:

```
echo 4194303 > /proc/sys/kernel/pid_max
```

(2) 预读设置

Linux 内核的预读功能可以有效地减少磁盘的寻道次数和应用程序的 I/O 等待时间, 是改进磁盘 I/O 性能的重要优化手段之一。而所谓的预读, 是指文件系统为应用程序一次读

① 另外两种处理器结构为多对称处理器结构 SMP 和海量并行处理结构 MPP。

② <http://lists.ceph.com/pipermail/ceph-users-ceph.com/2013-December/036211.html>

出比预期更多的文件内容并缓存在 page cache 中，这样应用程序的下一次请求将直接从 page cache 中读取，由于 page cache 较磁盘存储高效得多，因此预读机制可以提升应用程序的访问性能。对于 Ceph 集群，Linux 系统默认的预读 read_ahead_kb 并不适合读取 RADOS 对象数据，而是建议设置更大的值，根据 Ceph 公开的网上资料，8192 是比较合适的值，如下：

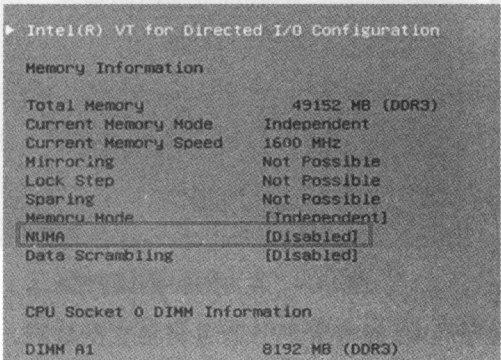


图 14-10 在 BIOS 中关闭 NUMA

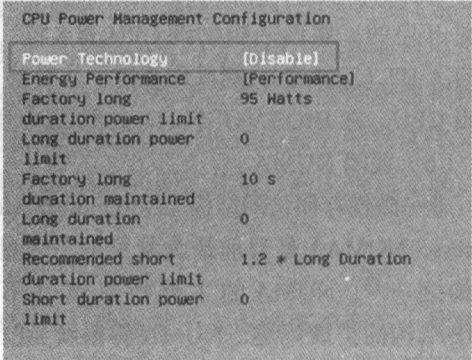


图 14-11 在 BIOS 中关闭节能模式

```
echo "8192" > /sys/block/sda/queue/read_ahead_kb
```

(3) I/O Scheduler

Linux 内核通常对 IO 调度进行了专门的优化设置，而在使用 SSD 的 Ceph 集群中，这种设置反而起到拖累作用，因此在 SSD 环境中建议使用 Noop 替换 CFQ 调度器，如下：

```
echo "noop" >/sys/block/sdX/queue/scheduler
```

其中，sdX 为系统中 SSD 硬盘对应的块设备名称，而如果 sdX 为 SAS/SATA 机械硬盘，则建议使用 deadline 调度器，如下：

```
echo "deadline" >/sys/block/sdX/queue/scheduler
```

(4) 交换空间 SWAP

交换空间本质上是外置硬盘上的存储空间，也称为虚拟内存。交换空间存在的目的在于防止服务器物理内存不够使用，当物理内存不够使用时，系统便将物理内存中的页面调度到外部硬盘中，此时便会产生大量 IO，通常也会伴随应用系统性能的下降。Linux 内核通过 vm.swappiness 内核参数控制 SWAP 策略，当 vm.swappiness 为 0 时，Linux 系统便会尽可能地避免发生 SWAP 操作，而当 vm.swappiness 为 100 时，Linux 系统便会积极进行 SWAP 操作，而如果存在足够物理内存的情况下也发生 SWAP 操作，则必然对系统性能造成比例影响，因此建议在物理内存充分的情况下设置 vm.swappiness 为 0，如下：

```
echo "vm.swappiness=0" >> /etc/sysctl.conf
```

(5) 内存管理 TCmalloc

Ceph 使用 TCmalloc 负责内存管理，TCmalloc 对 Ceph 性能的影响在社区和邮件队列

中有过很多讨论^①。其对 Ceph 性能造成影响的主要原因，在于 TCmalloc 的默认可用线程 Cache 太小（32MB），当 TCmalloc 进行繁重的多线程内存分配负载时，TCmalloc 会消耗大量的 CPU 资源。而增加 TCmalloc 的 Cache 大小需要设置系统环境变量，且建议设置为 256MB，如下：

```
TCMALLOC_MAX_TOTAL_THREAD_CACHE_BYTES=268435456
```

此外，还可以通过释放 TCmalloc 已分配但是 Ceph 进程实际上并未使用的系统内存，从而可在一定程度上增加系统性能。内存释放过程很简单，仅需一个命令即可完成，如下：

```
# ceph tell osd.* heap release
```

与 Ceph 默认的 TCmalloc 相比，JEmalloc 似乎是个很好的选择，公开的资料显示^②，当 Ceph 使用 JEmalloc 而非老版本的 TCmalloc 时，IOPS 性能提高了将近 4.7 倍，而在 4kb 的小 IO 随机写测试中，JEmalloc 比 TCmalloc 快进 4.21 倍，不过 JEmalloc 也消耗了更多的内存。由于 JEmalloc 并没有被打包到 Ceph 中，因此需要对 Ceph 重新编译打包才能使用，编译过程中修改如下选项：

```
--with-jemalloc
--without-tcmalloc
```

3. Ceph 配置优化

Ceph 的配置参数位于 /etc/ceph/ceph.conf 文件中，关于 Ceph 各个配置选项的解释说明和默认值请参考 14.1.2 节的内容，尽管 Ceph 使用默认配置参数也可正常启动，但是默认参数无法为 Ceph 带来最佳的性能体验，因此通常需要对 Ceph 各个配置参数进行重新设置。对于性能优化而言，不同用户环境中可能需要设置不同的参数值才能带来最佳的 Ceph 性能，而且性能调优也是个日积月累的经验总结过程，对于任一参数而言，没有绝对正确的值适合所有用户。下面是根据运维经验和网上公开的 Ceph 配置推荐汇总得到的 Ceph 配置参考，用户在部署 Ceph 时可以直接使用以下配置，也可以参考以下配置进行性能调优。

```
[global]
fsid = 4042b616-be34-44c1-b806-8a80f1fcd11d
mon host = 192.168.142.44,192.168.142.45,192.168.142.46
auth cluster required = cephx
auth service required = cephx
auth client required = cephx
osd pool default size = 3
osd pool default min size = 1
osd pool default pg num = 128
osd pool default pgp num = 128
public network = 192.168.115.0/24
cluster network = 192.168.142.0/24
```

① <https://www.mail-archive.com/search?l=ceph-users%40lists.ceph.com&q=tcmalloc&start=0>

② <http://www.sebastien-han.fr/blog/2015/09/07/the-ceph-and-tcmalloc-performance-story/>


```
max open files = 131072
mon initial members = controller1, controller2, controller3
```

```
[mon]
```

```
mon data = /var/lib/ceph/mon/ceph-$id
mon clock drift allowed = 1
mon osd min down reporters = 13
mon osd down out interval = 600
```

```
[osd]
```

```
osd data = /var/lib/ceph/osd/ceph-$id
osd journal size = 20000
osd journal = /var/lib/ceph/osd/$cluster-$id/journal
osd mkfs type = xfs
osd mkfs options xfs = -f -i size=2048
filestore xattr use omap = true
filestore min sync interval = 10
filestore max sync interval = 15
filestore queue max ops = 25000
filestore queue max bytes = 1048576000
filestore queue committing max ops = 50000
filestore queue committing max bytes = 10485760000
filestore split multiple = 8
filestore merge threshold = 40
filestore fd cache size = 1024
journal max write bytes = 1073714824
journal max write entries = 10000
journal queue max ops = 50000
journal queue max bytes = 10485760000
osd max write size = 512
osd client message size cap = 2147483648
osd deep scrub stride = 131072
osd op threads = 16
osd disk threads = 4
osd map cache size = 1024
osd map cache bl size = 128
osd mount options xfs = "rw,noexec,nodev,noatime,nodiratime,nobarrier"
osd recovery op priority = 2
osd recovery max active = 10
osd max backfills = 4
osd min pg log entries = 30000
osd max pg log entries = 100000
osd mon heartbeat interval = 40
ms dispatch throttle bytes = 1048576000
objecter inflight ops = 819200
osd op log threshold = 50
osd crush chooseleaf type = 0
osd crush update on start = false

[client]
rbd cache = true
rbd cache size = 335544320
```

```

rbd cache max dirty = 134217728
rbd cache max dirty age = 30
rbd cache writethrough until flush = false
rbd cache max dirty object = 2
rbd cache target dirty = 235544320

```

14.2 Ceph 运维与常见故障处理

14.2.1 Ceph OSD 与 PG 状态

在 Ceph 存储集群的日常运行维护中，OSD 和 PG 运行状态是诊断 Ceph 集群的首要参考，尤其是 PG 的众多状态，从各个角度反映了 Ceph 底层数据的存储、迁移和恢复等过程，因此在对 Ceph 存储集群进行健康检查和故障诊断前，必须熟悉并理解 OSD 和 PG 的各个状态所蕴含的 Ceph 后端正在进行的操作和状态的变更。

1. OSD 状态理解

相对于 PG，位于底层的对象存储设备 OSD 并无太多的运行状态可供参考（总共有 4 种状态），在一个正常运行的 Ceph 集群中，所有 OSD 应该全部处于 in 和 up 的状态，如下所示：

```

[root@controller1 ~]# ceph -s
cluster 4042b616-be34-44c1-b806-8a80f1fcd11d
health HEALTH_OK
monmap e1: 3 mons at {compute1=192.168.142.44:6789/0,compute2=192.168.142.45:6789/0,
storage1=192.168.142.46:6789/0}, election epoch 102, quorum 0,1,2 compute1,
compute2,storage1
osdmap e83: 5 osds: 5 up, 5 in
pgmap v819: 576 pgs, 6 pools, 147 MB data, 46 objects
        648 MB used, 45376 MB / 46025 MB avail
        576 active+clean

```

上述 Ceph 集群中共有 5 个 OSD，且集群处于 HEALTH_OK 状态，可以看到 OSDMAP 中记录了 5 个 OSD，并且 5 个 OSD 都处于 up 和 in 状态。除了 up 和 in，OSD 还有 down 和 out 状态，并且 up 与 down 是相对 OSD 进程而言的，out 与 in 是相对 Ceph 集群而言的，而 OSD 的状态通常由 OSD 进程状态和其相对于 Ceph 集群的状态组成，OSD 状态的变化如图 14-12 所示。

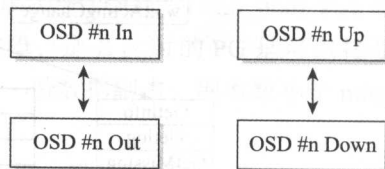


图 14-12 Ceph OSD 状态变化

在 OSD 的状态中，in 和 out 可以相互转换，而 up 和 down 也可以相互转换。下面对这几个 OSD 状态的含义进行解释。

□ up 且 in：up 且 in 是正常的 OSD 状态，此时说明该 OSD 进程正常运行，并且其上承载了至少一个 PG，即该 OSD 上已被分配了 POOL 的 PG。up 且 in 是理想的

OSD 状态。

- ❑ up 且 out: up 意味着此 OSD 进程正常运行，但是已被踢出 Ceph 集群，即不承载任何存储池的 PG，或者说此 OSD 上不会存储任何客户端数据。
- ❑ down 且 in: down 且 in 是异常的 OSD 状态，此时说明该 OSD 进程已出现异常，但是还未被踢出 Ceph 集群，即其上仍然承载着存储池 PG。OSD 异常后，根据用户设置的 `mon osd down out interval` 选项值（如 300s），Ceph 会在 OSD 异常后的 5 分钟后才将 OSD 标记为 out。
- ❑ down 且 out: down 且 out 说明此时该 OSD 已经完全脱离 Ceph 集群，已经不承载任何的存储池 PG，通常如果 OSD 对应的存储硬盘故障一段时间后，该 OSD 便会被 Ceph 标记为 down 且 out 状态。

2. PG 状态理解

PG 是 Ceph 存储集群中非常重要的概念，PG 通常也被称为放置组（Placement Group），是 Ceph 为了数据存储而引入的虚拟概念，并不对应具体的物理设备。当客户端数据存入 Ceph 集群时，数据首先被分割成为 Object，Object 再被映射到 PG，最终 PG 被映射到对象存储设备 OSD。关于 Ceph 数据存储原理和过程可参考本书第 10 章的相关章节。当管理员对 Ceph 集群进行健康检查时（`ceph -s` 或 `ceph -w`），Ceph 便会输出 PG 的状态，理想情况下，Ceph 的全部 PG 状态应该为 active+clean，不过除了 active 和 clean 外，PG 还有很多其他状态。PG 采用状态机（State Machine）的机制来管理众多状态，PG 的所有状态类似树形结构，每个状态可能存在子状态，而子状态又可能还存在更下一层的子状态，如图 14-13 所示。

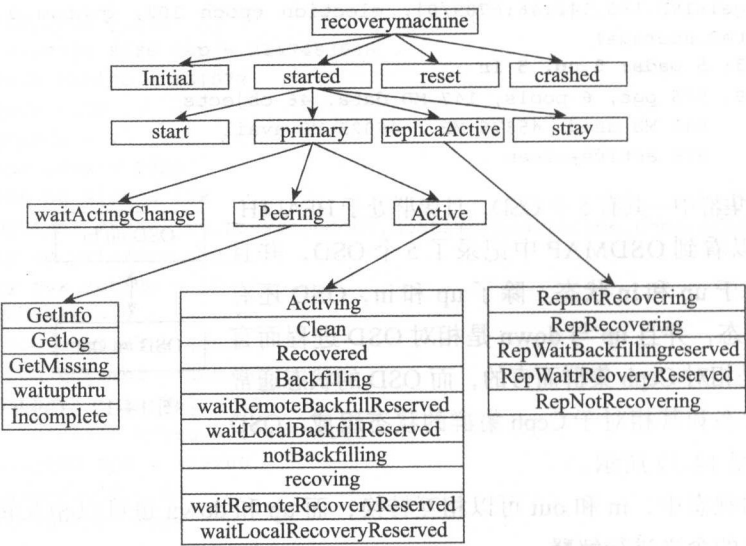


图 14-13 PG 状态构成的树形结构

在实际应用中，PG 的状态可能是图 14-13 中多个状态的组合，在 PG 状态机的管理下，

不同状态在接收到相应事件的驱动后进行彼此之间的转换。PG 状态的变化在其被创建或被扫描时便开始进行，从 PG 的初始状态到最终的数据同步完成的各过程中，PG 的状态机对这个过程中 PG 所经历各个状态进行标记和处理，并最终实现 active+clean 的 PG 状态。下面对实际应用中最常见和遇到的 PG 状态进行解释，通过这些状态的理解，便可在 Ceph 集群故障时，通过 PG 状态来推导问题所在。

- ❑ **creating**：PG 正在创建过程中，当用户创建 POOL 时，通常会根据配置文件 ceph.conf 中设置的 `osd pool default pg num` 或者创建 POOL 时指定的 PG 数目来创建 PG。PG 创建后，便开始一系列的状态转换，直至最终的 active+clean 状态。
- ❑ **peering**：PG 正在进行 peering 过程。peering 是 Ceph 集群一个非常重要的过程，在 PG 进行 peering 期间，客户端 IO 不能访问此 PG。peering 过程中，Acting set 中的 OSD 完成互联，并实现各个 OSD 中对象状态及其元数据的一致性，例如 POOL 中的数据副本数为 3，则该 POOL 中的一个 PG 会被映射给 3 个 OSD，在该 PG 状态为 peering 时，其所对应的 3 个 OSD 便进行互联和对象数据状态及其元数据的同步。peering 完成并不意味着 Acting set 中的 OSD 所存储的数据内容是同步或最新的，通常 peering 完成后会进行 recovery 或 backfilling 操作，此时各个 OSD 上的副本数据才会达到同步。
- ❑ **active/clean**：当 PG 为 active 时，表明此 PG 可以接受客户端访问，这是 PG 完成 peering 之后的状态。而 clean 状态则表明此 PG 中的对象已被成功复制了 N 份，N 为此 PG 所属存储池设定的数据副本数目。active 与 clean 状态的组合是 Ceph 集群最健康的状态。
- ❑ **degraded**：当 PG 中的对象还没有被复制正确的份数时，此 PG 便会处于降级 (degraded) 状态。假设 PG 所属 POOL 的 `pool_size` 为 3，即一个 PG 中的所有对象均会映射到三个 OSD，如果此时某个 OSD 故障，即此时 PG 中的数据对象仅有两份副本，而不是 POOL 规定的三份，则 PG 会被标记为 degraded。当 PG 处于 active+degraded 时，PG 是否可以接受客户端请求取决于 POOL 的 `min_pool_size` 参数，如果 `min_pool_size` 为 2，且仅有一个 OSD 故障，那么此时的 PG 是可以接受 IO 请求的，而如果有两个 OSD 故障，则此时仅有一份数据副本，副本数小于 `min_pool_size`，PG 将不能接受 IO 请求。
- ❑ **scrubbing**：Ceph 会定期对每个 PG 中的对象进行数据一致性的校验，类似 Linux 文件系统中的 fsck 功能。当 PG 处于 scrubbing 状态时，表明此 PG 正在被 scrub，默认情况下，Ceph 每天进行一次常规 scrub，每周进行一次深度 scrub，如果 scrub 操作发现 PG 的多份副本数据之间存在不一致的情况，则会触发 repair 或 recovery 操作使其恢复一致。
- ❑ **inconsistent**：当 Ceph 检测到 PG 中某个对象的一份或多份副本数据存在不一致情况时（如对象大小不一致或对象副本丢失等），PG 便会被标记为 inconsistent 状态，

当 PG 处于 inconsistent 状态时, 会自动对其进行修复, 修复中的 PG 将处于 repair 状态。

- ❑ recovering: 当 Ceph 正在迁移或同步 PG 对象副本时, 该 PG 将被标记为 recovering 状态。在实际运行中, 如果某个 OSD 故障之后又重新恢复到集群中, 而在此 OSD 故障期间 PG 数据可能已被更新, 则 OSD 上的数据就会落后于 PG 的其他 OSD 副本数据, 此时 Ceph 便会同步 PG 的对象副本数据, 使其全部对象副本均达到最新状态, 而此时的 PG 便处于 recovering 状态。
- ❑ backfilling: backfill 和 recovery 是 Ceph 集群内部进行数据转移的两种主要形式, recovery 主要发生在原有 OSD 数据不能反映最近更新数据的时候, 而 backfill 则是向空白的 OSD 中填入 PG 的对象副本, backfill 是 recovery 的一种特例。当 Ceph 集群有新的 OSD 加入时, Ceph 会重新分布 PG, 重新分布的过程就是将 PG 的某个副本数据由原来的 OSD 全部重新转移到新的 OSD 上, 这个过程中 PG 会被标记为 backfilling 状态。
- ❑ remapped: 当 PG 的 Acting set 发生变化时, 数据会由原 OSD 集合迁移至新的 OSD 集合, 在迁移过程中 PG 会被临时映射到 CRUSH 指定的 OSD 上, 待迁移完成后 PG 才会映射到新的 Acting set 中, 在数据迁移过程中 PG 便会被标记为 remapped 状态。
- ❑ incomplete: 当 Ceph 发现 PG 中的某个写操作可能已经完成, 但却丢失了某些信息或者 PG 存在不健康的数据副本时, PG 将被标记为 incomplete 状态, 当 PG 处于 incomplete 状态时, 通过重启某个故障的 OSD 进程即可解决。
- ❑ undersized: 当 PG 处于 undersized 状态时, 表明 PG 的数据副本数目没有达到 POOL 规定的副本数。
- ❑ peered: PG 的 peered 状态表明 PG 已经完成了 peer 过程, 即 PG 对应的 Acting set 中的 OSD 已经完成互联通信, 同时 Acting set 中各个 OSD 的对象状态和元数据信息已经同步, 但是 PG 的对象数据副本数目没有达到 POOL 规定的最小副本数, 即 min_pool_size 参数值规定的副本数, 此时 PG 不会接受客户端请求。
- ❑ Stale: Stale 表明 PG 处于一种未知的状态, Ceph 的 MON 进程不能从 PG Map 中获取与此 PG 相关的有效信息。

正常情况下, Ceph 中全部 PG 的状态应该为 active+clean, 如下所示:

```
[root@controller1 ~]# ceph pg stat
v819: 576 pgs: 576 active+clean; 147 MB data, 648 MB used, 45376 MB / 46025 MB avail
```

在对 Ceph 进行健康检查时, 如果发现 Ceph 中有状态不为 active+clean 的 PG 存在, 则需根据 PG 的具体状态值对其进行具体的分析, 下述是一个处于 HEALTH_WARN 状态的 Ceph 集群, PG 存在的各种状态如下所示:

```
[root@controller1 ~]# ceph -s
```

```
cluster 4042b616-be34-44c1-b806-8a80f1fcd11d
health HEALTH_WARN
24 pgs backfill //有PG属于回填中，据此可判断可能有新的OSD加入集群
58 pgs degraded //有PG降级，据此可判断可能有OSD故障发生
16 pgs recovering //有PG处于数据恢复中
54 pgs recovery_wait //有PG处于恢复等待中
65 pgs stuck unclean //有65个PG处于unclean状态
30 pgs undersized //有30个PG的副本数目小于POOL规定的副本数，即pool size设定值
recovery4520 /28129objects degraded (16.069%)
```

通常而言，Ceph 的 PG 状态未处于 active+clean 并不意味着 PG 或 OSD 一定出现了问题，Ceph 的自我愈合能力会自动将 PG 带到 active+clean 的健康状态，但是当 PG 处于 Stuck 时，Ceph 的自我修复能力可能不起作用。Stuck 主要包括 Unclean、Inactive 和 Stale 三种状态，其中 Unclean 意味着 PG 中的对象副本还未达到规定的副本数，此时的 PG 应该处于恢复中，而 Inactive 状态时 PG 不能响应任何客户端的读写请求，此时的 PG 正在等待存储了最新数据的 OSD 恢复，Stale 则说明 PG 处于未知状态，因为映射了此 PG 的 OSD 有一段时间未向集群监控报告状态。如果要查看集群中处于 Stuck 的 PG，可通过如下命令查看：

```
ceph pg dump_stuck unclean|inactive|stale
```

14.2.2 Ceph OSD 节点增删

随着业务系统的增加和业务数据的递增，任何存储系统都会面临扩容的问题。在 Ceph 存储集群中，存储扩容实际上就是增加集群 OSD，扩容过程可以是在原有存储节点上增加 OSD，也可以是新增存储节点的同时增加 OSD，通常在 Ceph 集群初始化时，存储节点上的磁盘驱动已被全部用于 OSD，因此 Ceph 集群的扩容通常意味着新增存储节点。对 Ceph 而言，日常检查中观察记录集群剩余可用空间是十分必要的，如果 Ceph 存储空间接近 nearfull ratio 设定的比例值，就必须考虑扩容 Ceph 容量，因为此时如果有 OSD 故障，则 Ceph 的使用空间很可能超出 full ratio 而禁止一切客户端访问。

Ceph 最突出的功能之一便是允许通过增删 OSD 节点形式动态增加和减少集群容量，但是如果在集群容量增减之前不进行仔细规划，则可能对客户端造成灾难性的影响。在 Ceph 集群中，当 OSD 数目出现变化时，集群 OSDMap 和 CRUSHMap 也会发生变化，此时集群中的 PG 便会根据新的 CRUSHMap 重新计算 Primary 和 Replicas OSD，进而引起 PG 数据向新加入的 OSD 迁移（Ceph 数据的 Reblance 过程），即发生集群数据回填（Backfill）和恢复（Recovery）操作。由于 Backfill 和 Recovery 操作对集群性能有很大影响，因此如果此时 Ceph 客户端正在进行大量的 IO 操作，而且有大量的 OSD 同时加入或退出集群，则客户端 IO 操作必然受到极大影响甚至出现响应超时异常。所以，在对生产环境中的 Ceph 集群进行容量增减之前，务必进行详细的规划和谨慎操作，下面给出几个增减 Ceph 容量时应该注意的参考建议：

（1）合理设置容量空间阈值

Ceph 提供了 `near full ratio` 和 `full ratio` 参数来进行容量阈值的设置, 当 Ceph 的使用空间达到 `full ratio` 时, 为了防止数据丢失, Ceph 将停止一切对外服务, 因此 Ceph 还提供了 `near full ratio` 参数来向管理员发出容量告警。当 Ceph 使用空间接近 `near full ratio` 时, 管理员必须准备扩容集群容量, 否则将使 Ceph 集群面临不可访问的风险。Ceph 默认的 `near full ratio` 和 `full ratio` 值分别为 85% 和 95%, 但是在生产环境中, 建议根据实际情况减少这两个值 (如 75% 和 85%), 如此, 便可在 Ceph 出现容量告警时为自己留下更多的扩容响应时间。此外, 如果是删除 OSD, 则操作之前需计算清楚删除 OSD 后剩余容量是否会达到 `full ratio`, 否则删除 OSD 后 Ceph 集群可能无法使用。

(2) 防止骤增或骤减容量

在对 Ceph 进行动态增减容量时, 应尽可能地遵循“勤拿少取”的原则。由于增删 OSD 会触发 Ceph 集群的 `Rebalance` 操作, 而增删的 OSD 越多, `Backfill` 和 `Recovery` 操作所需的时间就越长, 因此 `Rebalance` 过程对 Ceph 集群性能的影响就越持久, 例如一次性增删 10 个 OSD 对 Ceph 性能的影响要远远大于仅增删 1 个 OSD 的影响。要防止骤增骤减 OSD, 最有效的方式便是设置较低的 `near full ratio`, 以便尽早告警并尽早扩容 Ceph 空间, 这样便有足够的缓冲时间进行逐个 OSD 扩容。

(3) 选择合理的扩容时间窗口

扩容 Ceph 会引起集群 `Rebalance`, 从而影响 Ceph 性能, 尤其是在一次性增加大量 OSD 时, Ceph 性能将受到极大影响。因此, 为了以防万一, 对生产系统中的 Ceph 集群进行扩容的时间窗口务必合理选择, 例如在业务低峰时段的夜晚或凌晨。

(4) 临时禁用 Scrub 操作

`Scrubbing` 操作在保证数据一致性和持久性方面是十分必要的, 但是 `Scrubbing` 也是十分消耗资源的操作。因此, 在对 Ceph 集群进行增删 OSD 时, 最好禁止 Ceph 集群的 `Scrubbing` 和 `DeepScrubbing` 操作, 同时等待当前的 `Scrubbing` 操作完成。要禁止 `Scrubbing` 操作, 可参考如下命令:

```
ceph osd set noscrub
ceph osd set nodeep-scrub
```

在完成对 Ceph 集群 OSD 的增删操作, 且集群处于 `active+clean` 状态之后, 再重新使用 Ceph 集群的 `Scrubbing` 操作, 如下:

```
ceph osd unset noscrub
ceph osd unset nodeep-scrub
```

(5) 限制 Backfill 和 Recovery 操作

在增删 Ceph 集群的 OSD 后, Ceph 通过 `Backfill` 和 `Recovery` 操作进行数据 `Rebalance` 操作, 而 Ceph 提供了控制 `Backfill` 和 `Recovery` 操作的配置选项, 通过这些配置选项可以控制 `Backfill` 和 `Recovery` 对 Ceph 性能的影响, 通常而言, 对 Ceph 性能影响越小, 则完成 `Backfill` 和 `Recovery` 所需时间便越长, 而在客户端高 IO 的应用场景下, 通常是宁可缓慢进

行 Backfill 和 Recovery, 也不希望客户端 IO 受到明显影响。更何况, 只要数据副本数设置合理 (如 `osd pool default size = 3` 并且 `osd pool default min size = 2`), 则 Ceph 在 degraded 状态下也可正常访问, 因此只要 Backfill 和 Recovery 不会太影响性能, 其在后台慢慢执行也无妨。要控制 Backfill 和 Recovery 对 Ceph 性能的影响, 可参考如下配置:

```
osdmaxbackfills = 1
osdrecoverymaxactive = 1
osdrecoverypriority = 1
```

此外, 用户还可通过 `osdrecoverysleep` 选项控制 Recovery 操作进行的时间, 以便使其在可预知的业务低峰时段进行, 从而最大限度地减少数据恢复操作对 Ceph 性能的影响。

(6) 扩容期间禁止 Backfill 和 Recovery 操作

限制 Backfill 和 Recovery 操作并不等于不进行 Backfill 和 Recovery 操作, 在逐个增加 OSD 过程中, 为了防止 Ceph 来回迁移数据, 可以在增删 OSD 之前, 先禁止 Ceph 的 Backfill 和 Recovery 操作, 同时为了防止在操作期间有 OSD 出现故障而被标记为 out 状态而导致数据迁移发生, 可以先禁用 Ceph 集群的 out 标记, 如下:

```
ceph osd set noout
ceph osd set nobackfill
ceph osd set norecovery
```

待 OSD 逐个添加完成, 同时 PG 的 Remap 计算完成后, 才重新开启 Backfill 和 Recovery 操作, 此时 Ceph 便开始 Rebalance 操作, 如下:

```
ceph osd unset noout
ceph osd unset nobackfill
ceph osd unset norecovery
```

1. 增加 OSD 节点

(1) 主机配置准备

主机硬件资源配置可参考 14.1.1 和 14.1.5 节, 此处需要注意的是主机硬盘配置, 由于技术的飞速发展, 硬盘容量也在不断增加, 因此, 扩容主机的单盘容量可能远大于初期上线时的硬盘容量, 而 Ceph 采用数据均匀分布方式, 如果新硬盘容量大于原硬盘, 则需要调整对应新硬盘的 OSD 权重。根据最佳实践, 扩容的新硬盘最好在型号容量上与原硬盘保持一致。

(2) 扩容主机上安装 Ceph 软件

主机系统配置完成后, 在 Ceph-admin 节点的 `hosts` 文件中增加扩容主机的主机名, 并为扩容主机配置 SSH 无密码访问, 之后通过 Ceph-deploy 工具安装部署 Ceph 软件包, 安装命令如下:

```
ceph-deploy install {ceph-node} [{ceph-node}...]
```

(3) 扩容主机上创建 OSD

首先查看主机上有哪些硬盘可用来创建 OSD，命令如下：

```
ceph-deploy disk list {node-name}
```

初始化扩容节点上磁盘，命令如下：

```
ceph-deploy disk zap {node-name}:{disk-name}
```

准备扩容节点 OSD，命令如下：

```
ceph-deploy osd prepare {node-name}:{data-disk}[:{journal-disk}]
```

激活扩容节点上的 OSD，命令如下：

```
ceph-deploy osd activate {node-name}:{data-disk}[:{journal-disk}]
```

(4) 更新 Ceph 的 CRUSHMap 信息

CRUSHMap 信息可以通过两种方式更新，一种为反编译方式，即 14.1.3 节中介绍的方式，另一种为命令行方式。使用命令行方式虽然简单，但是也需要掌握 Ceph 的 Bucket 层级结构，OSD 节点属于 Host 层级的 Bucket 且属于 Default Bucket 的项目，而 OSD 属于 Host Bucket 的项目，如果 CRUSH 采用 Ceph 默认的主机层级故障域，则新增 OSD 节点后可按如下命令行方式更改 CRUSHMap：

```
ceph osd crush add-bucket {node-name} host //添加host层级的Bucket，Bucket名称是扩容节点主机名
ceph osd crush move {node-name} root=default //将新增的host Bucket移到root层级Bucket中
ceph osd crush add {osd-name}{weight} host={node-name} //将新增OSD移到host层级中
```

(5) 为扩容节点新增下一个 OSD

为了不过多影响 Ceph 性能，建议逐个新增 OSD，即待上一个新增 OSD 完成 Backfill 和 Recovery，并实现 active+clean 状态后，再重复新增下一个 OSD，操作过程只需重复上述步骤即可。

2. 删除 OSD 节点

(1) 停止欲删除的 OSD 进程

删除 OSD 之前，请务必确保 OSD 删除后 Ceph 剩余容量未接近 near full ratio 或者 full ratio，否则删除正常 OSD 后 Ceph 将不能正常使用。停止 OSD 进程命令如下：

```
stop ceph-osd id={osd-num} //停止OSD进程
```

(2) 从 CRUSHMap 移除 OSD

注意，一旦执行此操作，Ceph 将触发 Rebalance 操作，客户端 IO 可能受到影响，操作之前请仔细评估。移除 OSD 的命令如下：

```
ceph osd out osd.{osd-num} //将OSD标记为out
ceph osd crush remove osd.{osd-num} //将OSD从CRUSH Map中删除
```

(3) 清除 OSD 认证信息

OSD 被移除后，认证信息不会自动删除，删除 OSD 认证信息的命令如下：

```
ceph auth del osd.{osd-num} //删除OSD的认证信息
```

(4) 移除 OSD

现在可以彻底删除 OSD，命令如下：

```
ceph osd rm {osd-num} //删除OSD
```

14.2.3 Ceph MON 节点增删

Ceph 通过轻量级 Monitor 进程维护主副本集群 Map 信息，Ceph 客户端通过 Monitor 进程获取集群 Map 副本信息从而将客户端绑定到特定的 POOL 中并进行 I/O 操作。Ceph 支持运行时动态增加或删除 Monitor 节点，理论而言，Ceph 仅需一个 Monitor 节点即可正常运行，但是对于生产环境，建议至少部署三个 Monitor 节点。此外，Ceph 使用改进后的 Paxos 协议为集群中的 Map 和其他关键信息建立共识，而鉴于 Paxos 的特性，Ceph 需要同时运行多个 Monitor 进程以便建立 Quorum 仲裁机制。在生产环境中，建议部署奇数个 Monitor 节点，因为奇数 Monitor 节点数目要比偶数 Monitor 节点具有更高的可用性，例如为了保证 Monitor 进程的仲裁机制，在两个 Monitor 节点的场景中，Ceph 无法容忍任何节点故障，在三个 Monitor 节点场景中，可以容忍一个节点故障，四个 Monitor 节点场景中，同样只能容忍一个节点故障，而在五个 Monitor 节点场景中，可容忍两个节点故障，这也是为何推荐部署奇数个 Monitor 节点的主要原因。归纳而言，Ceph 要求集群内全部 Monitor 节点中半数以上的 Monitor 节点处于运行状态同时彼此之间可以正常通信，“半数以上”的 Monitor 节点处于运行状态，可以是单个 Monitor 节点，或者两个 Monitor 节点、或者三个中的两个 Monitor 节点又或者四个中的三个 Monitor 节点等。对于多节点 Ceph 集群部署，建议至少部署三个 Monitor 节点，同时如果确实需要增加 Monitor 节点数目，建议一次性增加两个，从而保证 Monitor 节点数目以奇数形式递增。此外，由于 Monitor 是轻量级进程，因此也可以将 Monitor 进程部署到 OSD 节点上，不过鉴于与 Kernel 的 fsync 问题可能影响 OSD 节点性能，从最佳实践角度考虑最好将其独立部署。

1. 增加 Monitor 节点

(1) 主机配置准备

主机硬件资源配置可参考 14.1.1 和 14.1.5 节，需要指出的是，尽管轻量级的 Monitor 进程可以重叠部署在运行其他应用程序（如 OSD）的服务器上，但是如果有条件仍然建议独立部署 Monitor 进程。操作系统配置方面，需预先配置好 NTP、yum 源和网络及其安全组设置等。

(2) Ceph 相关软件包安装

在新增的 Monitor 节点上安装 Ceph 软件，准备好 Ceph 相关软件包的 yum 源后，可以

通过 yum 或 ceph-deploy 安装 Ceph，如下：

```
yum install -y ceph //yum命令行安装
ceph-deploy install ceph <hostname> //ceph-deploy工具安装
```

(3) 编辑 Ceph 配置文件

在 Ceph-admin 节点编辑 Ceph 配置文件 ceph.conf，添加新增 Monitor 节点至初始 Quorum 列表中（可选步骤），如下：

```
[global]
mon_initial_members = node0, node1, <hostname>
```

添加 Monitor 节点的监控 IP 地址，如下：

```
[mon.<hostname>]
public_addr = <ip-address>
```

(4) 推送配置文件至集群节点

Ceph 集群的配置文件变更后，需要重新将其推送至全部 Ceph 节点，如下：

```
ceph-deploy --overwrite-conf config push <ceph-node0 ceph-node1 ...>
```

(5) 将 Monitor 节点添加至集群中

添加 Monitor 节点至 Ceph 集群中，如下：

```
ceph-deploy mon add <hostname>
```

(6) 检查监控是否已加入集群

Monitor 节点添加完成后，检查是否已添加成功，如下：

```
ceph quorum_status --format json-pretty
```

2. 删除 Monitor 节点

Monitor 节点是 Ceph 集群的协调中心，没有特殊情况不要轻易删除 Monitor 节点，如果确实要减少集群中的 Monitor 节点数目，一定要预留足够的 Monitor 节点数目以保证 Ceph 监控节点的 Quorum 机制。Monitor 节点的删除过程可参考如下步骤实现：

(1) 关闭 Monitor 节点 MON 进程

确定要删除的 Monitor 节点后，将其上的 MON 进程关闭，MON 进程关闭后，查看集群是否重新选举生成新的 Leader MON，下述 Ceph 集群中，Leader MON 为 compute1 节点：

```
[root@controller1 ~]# ceph quorum_status --format json-pretty
{
  "election_epoch": 166,
  "quorum": [
    0,
    1,
    2],
  "quorum_names": [
    "compute1",
```

```

    "compute2",
    "storage1"],
    "quorum_leader_name": "compute1",
    "monmap": { "epoch": 1,
                  "fsid": "4042b616-be34-44c1-b806-8a80f1fcd11d",
                  "modified": "0.000000",
                  "created": "0.000000",
                  "mons": [
                    { "rank": 0,
                      "name": "compute1",
                      "addr": "192.168.142.44:6789\0"},
                    { "rank": 1,
                      "name": "compute2",
                      "addr": "192.168.142.45:6789\0"},
                    { "rank": 2,
                      "name": "storage1",
                      "addr": "192.168.142.46:6789\0"}]}}}

```

待新的 Leader MON 生成后，进行下一个步骤。

(2) 将 Monitor 节点从集群中删除

待删除节点上的 MON 进程关闭且新的 Leader MON 生成后，将 Monitor 节点从集群中删除，Monitor 节点被删除后，MONMap 中将不再存在此节点的任何信息，删除命令如下：

```
ceph mon remove <hostname>
```

(3) 更新 Ceph 配置文件

Monitor 节点删除后，在 Ceph 配置文件 `ceph.conf` 中删除与其相关的所有条目。

(4) 推送更新后的配置文件

Ceph 配置文件更新之后，重新将其推送至 Ceph 集群中，如下：

```
ceph-deploy --overwrite-conf config push <ceph-node0 ceph-node1 ...>
```

(5) 使用 ceph-deploy 删除 Monitor 节点

为了确保完全删除 Monitor 节点，使用 `ceph-deploy` 工具再次删除，如下：

```
ceph-deploy mon destroy <hostname>
```

(6) 检查 Monitor 节点是否已被成功删除

Monitor 节点删除完成后，检查是否已成功删除，如下：

```
ceph quorum_status --format json-pretty
```

14.2.4 Ceph Journal 故障维护

Ceph 采用日志机制以保证数据副本的一致性和小块随机 IO 的快速提交，当 Ceph 客户端向 POOL 中的 PG 写入数据时，Ceph 根据 CRUSH Map 信息将 PG 数据副本映射到 OSD，而数据在写入 OSD 时首先会被写入与之对应的日志盘（Journal Disk）中，随后日志中的数

据才同步到 OSD 中，因此，如果与 OSD 对应的日志分区或磁盘出现故障时，所有与之关联的 OSD 都将不能正常工作。在 Ceph 集群的日常运维中，与日志相关的两个常见操作主要是正常日志盘的更换（如 SATA 盘替换为 SSD 盘或者由分区替换为整盘）和故障日志盘的更换。

1. 正常日志盘替换

在 Ceph 集群正常运行时，可能希望将现有 OSD 的日志盘进行替换，如以前可能是多个 OSD 进程通过磁盘分区形式同时写一个 SSD 磁盘，但是基于性能考虑，现在希望 OSD 写入独立的日志盘，或者以前的日志盘为 SAS/SATA 磁盘，同样基于性能考虑，现在希望替换为 SSD 固态硬盘。要替换 OSD 的日志盘，可参考如下过程实现：

(1) 确认与日志盘相关的 OSD

OSD 与 Journal 密切相关，任何对 Journal 的操作都会影响到与之相关的 OSD，因此在操作 Journal 之前务必确认清楚与其相关的全部 OSD。正常情况下，OSD 的默认日志及其大小如下：

```
osd journal = /var/lib/ceph/osd/$cluster-$id/journal
osd journal size = 5120
```

此外，可通过命令行方式来验证 OSD 的日志路径，以 osd.0 为例，如下：

```
[root@compute1~]# ceph --admin-daemon /var/run/ceph/ceph-osd.0.asok config show
| grep\
osd_journal
"osd_journal": "\var\lib\ceph\osd\ceph-0\journal",
"osd_journal_size": "5120",
[root@compute1 ceph-0]# cd/var/lib/ceph/osd/ceph-0
[root@compute1 ceph-0]#ls -l
total 56
.....
lrwxrwxrwx  1 root root      9 Oct 16 00:44 journal -> /dev/sdb1
.....
[root@compute1 ceph-0]# pwd
/var/lib/ceph/osd/ceph-0
```

可以看到，OSD 进程 osd.0 的日志为 /var/lib/ceph/osd/ceph-0/journal，其中“ceph-0”的“ceph”为集群名称，而“0”为 OSD 的 ID 编号。此外，osd.0 的日志大小为 5120MB，其对应的磁盘分区为 /dev/sdb1。

(2) 为 OSD 设置 noout 标志

因为操作 Journal 需要停止与之关联的 OSD，而 Ceph 在一段时间后（默认 300s）会自动将 down 状态的 OSD 标记为 out 状态进而触发数据恢复操作，而数据恢复操作对集群性能有影响，因此最好暂时关闭 out 标记，如下：

```
ceph osd set noout
```

(3) 关闭 OSD 进程

关闭与 Journal 相关的全部 OSD 进程，以 osd.0 为例，如下：

```
stop ceph-osd id=0
```

(4) 将日志盘中的数据刷入 OSD

因为要替换 OSD 日志盘，而正常情况下 OSD 中可能还有数据未同步到 OSD 中，因此手工将其刷入 OSD，以 osd.0 为例，如下：

```
ceph-osd --flush-journal-i0 //0为osd.0的id
```

(5) 创建日志连接

准备好 Journal 分区，如 /dev/sde（或 /dev/sde1），删除 OSD 的原 Journal 连接，为 OSD 新建指向新 Journal 的连接，以 osd.0 为例，如下：

```
rm/var/lib/ceph/osd/ceph-0/journal //删除原osd的journal连接
ln-s /dev/sde1/var/lib/ceph/osd/ceph-0/journal //新建osd的journal连接
```

(6) 修改 Ceph 配置文件

在 ceph.conf 中将新的 Journal 添加到对应的 OSD 配置文件中，以 osd.0 为例，如下：

```
.....
[osd.0]
osd journal = /dev/sde1
osd journal size = 5120
.....
```

(7) 初始化 Journal 并启动 OSD

Journal 设置完成后，对其进行初始化并重新启动之前停止的 OSD 进程，以 osd.0 为例，如下：

```
ceph-osd--mkjournal-i0
startceph-osd id=0
```

(8) 取消 noout 标志

```
ceph osd unset noout
```

2. 故障日志盘更换

当 Journal 盘出现硬件故障时，与其关联的全部 OSD 都将无法工作，此时必须及时更换日志盘，以便 Ceph 集群尽快恢复 active+clean 状态。硬盘故障可通过 smartctl、dmesg 和 syslog 等信息进行判断，如果确实发现作为 Journal 的硬盘出现故障，则可通过以下步骤更换日志盘并恢复 OSD 进程：

(1) 更换故障硬盘

日志盘故障时，通常 OSD 已经处于 down 且 out 状态，如果 OSD 仍然 up，则参考前文关闭 OSD 进程，否则定位硬盘物理位置后，可直接用新盘将其更换。

(2) 新硬盘格式化分区处理

系统识别到新硬盘后,参考 Ceph 部署文档对作为 Journal 的新硬盘进行格式化分区处理。为了简化工作量,对新磁盘的格式化分区最好与之前的故障日志盘保持一致。

(3) 初始化受影响 OSD 的 Journal

为受影响的 OSD 重新初始化 Journal,如果有多个 OSD 同时受到影响,则依次为其初始化 Journal,以 osd.0 为例,如下:

```
ceph-osd--mkjournal-i0 //0为osd.0的id
```

(4) 启动受影响 OSD 进程

```
startceph-osd id=0 //0为osd.0的id
```

(5) 检查 Ceph 集群健康情况

当 OSD 启动后,由于其上数据落后于其他副本,通常会触发 Ceph 的 Recovery 操作,如果受影响 OSD 较多而且时间较长,则 Ceph 恢复所需的时间也较长,集群性能也会受到一定程度的影响,在此期间可通过 `ceph -s` 或 `ceph -w` 监控集群健康情况。

14.2.5 Ceph OSD 故障硬盘更换

在 Ceph 存储集群的日常维护中,最为常见的操作便是更换故障 OSD 硬盘。由于 Ceph 采用大规模廉价设备来提供高可用、高性能和高扩展性的存储集群,因此其硬盘或节点故障率相对昂贵的企业级存储设备会高出很多,这便要求存储管理员经常性地检查 Ceph 集群健康状况,一旦出现硬盘故障,需要尽快对其进行更换。下面介绍 OSD 硬盘故障时的更换修复过程,如下:

(1) 关闭 Ceph 集群的数据迁移

一旦硬盘故障导致 OSD 进程受影响,则其将处于 down 状态,在经过 `mon osd down out interval` 设定的时间间隔后,Ceph 将其标记为 out 状态,并开始进行数据迁移恢复。为了降低 Ceph 进行数据恢复或 Scrub 等操作对性能的影响,可以先将其暂时关闭,待到硬盘更换完成且 OSD 恢复后再开启,如下:

```
for i in noout nobackfill norecover noscrub nodeep-scrub;do ceph osd set $i;done
```

(2) 定位故障 OSD

确定受影响的 OSD 进程,如下:

```
ceph osd tree |grep -i down
```

(3) 卸载 OSD 的挂载目录

进入硬盘故障 OSD 节点,卸载 OSD 挂载目录,以 osd.0 为例,如下:

```
umount /var/lib/ceph/osd/ceph-0 //集群名称为ceph,osd的id为0
```

(4) 从 CRUSH Map 中移除 OSD

```
ceph osd crush remove osd.0 //将故障OSD(osd.0)从CRUSH Map中移除
```

(5) 删除故障 OSD 的秘钥

```
ceph auth del osd.0 //将故障OSD(osd.0)的秘钥删除
```

(6) 更换物理硬盘

定位故障物理硬盘，将其拔出并插入新磁盘，为了保证一致性，最好插入相同容量、相同型号的硬盘。

(7) 为新硬盘创建 OSD

系统识别到新硬盘后，注意识别出新更换的硬盘盘符（如 /dev/sdf），在有众多磁盘的 OSD 存储节点中，千万不要错误识别成其他正常运行的磁盘，否则其将被格式化，这可能带来意想不到的灾难性后果。此外，在运行 `ceph-deploy` 命令添加 OSD 时，务必确保先前的故障 OSD 已被彻底删除，否则可能添加与原 OSD 的 ID 不一致的新 ID，`ceph-deploy` 添加 OSD 命令如下：

```
ceph-deploy --overwrite-conf osd create --zap-disk {hostname}:sdf
```

(8) 重新启动集群禁用标志

待新添加的 OSD 进入 CRUSH Map 后，重新启动之前设置的禁用标记，如下：

```
for i in noout nobackfill norecover noscrub nodeep-scrub;do ceph osd unset $i;done
```

一旦新的 OSD 加入集群，Ceph 将进行 Backfill 和 Recovery 操作，为了降低数据迁移过程对客户端 I/O 的影响，可以为新加入的 OSD 设置权重为 0，之后再逐步增加至与其他 OSD 相同的权重值，以 `osd.0` 为例，OSD 权重设置命令如下：

```
ceph osd crush reweight osd.0 0
```

(9) 检查集群健康情况

新 OSD 加入集群后，集群通常需要一段数据迁移时间才能恢复 `active+clean` 状态，在这期间应密切关注客户端 I/O 性能和集群健康情况。

14.2.6 Ceph 存储节点停机维护

在日常维护中，可能出现 OSD 存储节点需要停机维护的情况，例如对性能不足的 OSD 存储节点扩容 CPU、内存或者更换高带宽网卡等，都要求对服务器进行下电操作，本节对正常运行中的 OSD 存储节点进行停机维护的过程进行梳理，描述的 OSD 存储节点仅运行 OSD 进程，对于某些将 OSD 存储节点同时作为 Monitor 节点或者虚拟机计算节点（如 OpenStack 计算节点）的用户，请在本节操作前参考本书相关内容正确停止 MON 进程或将节点上的虚拟机迁移到其他节点。

(1) 可行性评估

Ceph 的 OSD 存储节点上通常运行有大量的 OSD 进程，如果贸然停止存储节点，很可

能导致 Ceph 存储集群不可访问,因此在操作前务必进行可行性评估。主机维护首要考虑的便是故障域,例如 Host 级别故障域允许某台主机断电,Rack 级别故障域允许整个机柜断电,ROW 级别故障域允许整排机柜断电,而 ROOM 级别故障域允许整个机房断电,Region 故障域则允许某个地区的数据中心因自然灾害等原因不可使用。Ceph 默认采用 Host 层级故障域,即 PG 的多份数据副本被分布存储在不同主机上,因此只要 PG 存在两份以上的数据副本,任一存储节点不能使用并不会导致 Ceph 数据丢失,但是 Ceph 是否可访问则需谨慎考虑,例如某个生产存储池的 pool size 为 3, min pool size 为 2,如果存储节点停机维护使得存储池中 PG 的数据副本为 1 (小于 2),则该存储池将不能被客户端使用,而如果将 min pool size 设为 1,则在停机维护期间仍可继续访问 Ceph 集群。因此,在停机维护前,务必清楚当前 Ceph 集群使用的故障域级别,以及 Ceph 集群中各个存储池的数据副本数及其允许正常工作的最小副本数,数据副本数可通过如下命令查看:

```
[root@controller1 ~]# ceph osd pool get volumes min_size //POOL名称为volumes
min_size: 2
[root@controller1 ~]# ceph osd pool get volumes size
size: 3
```

(2) 临时禁止 Ceph 进行数据迁移操作

由于存储节点 OSD 在主机维护完成后即将返回集群,为了防止存储节点维护期间不必要的数据迁移,暂时关闭 Ceph 集群的数据迁移功能,在此期间 Ceph 集群将处于 degraded 状态,但是只要满足 min pool size 设定值,Ceph 正常访问则不会受到影响,而且因内部未进行数据 Rebalance 操作,性能也不会受到影响。进行如下设置即可关闭 OSD 的数据迁移:

```
for i in noout nobackfill norecover noscrub nodeep-scrub;do ceph osd set $i;done
```

(3) 停止维护节点 OSD 进程

确认维护的 OSD 节点后,参考 14.2.2 节停止 OSD 进程,这里无须将 OSD 从 CRUSH Map 中删除,也无须删除 OSD 的认证信息,因为此处停止的 OSD 在维护完成后需要重新启动。

(4) 关闭节点进行硬件维护

(5) 启动维护节点 OSD 进程

维护完成后,重新启动 OSD 此处节点上的全部 OSD 进程,确保重新启动后全部 OSD 处于 up 且 in 的状态。

(6) 开始数据同步

维护节点 OSD 进程恢复后,重新设置 OSD 锁定状态使其可进行数据恢复操作,由于 Recovery 操作对 Ceph 集群有很大性能影响,因此最好选择夜间等业务低峰期进行 Recovery 操作,如下:

```
for i in noout nobackfill norecover noscrub nodeep-scrub;do ceph osd unset $i;done
```

(7) 监控集群监控情况

由于 OSD 存储节点维护期间，客户端可能已经更新 PG 数据副本，因此重新回到集群中的 OSD 数据可能已经落后于其他副本 OSD 上的数据，此时 Ceph 会通过 Recovery 操作来同步新旧副本数据，同步时间与 OSD 存储节点的维护时间、受影响的 OSD 数目以及客户端 IO 负载情况有关，通常维护期间客户端 IO 负载越大、维护时间越长和受影响 OSD 数目越多，则同步过程所需时间越长，对 Ceph 性能的影响也越大，数据同步完成后 Ceph 将恢复到 active+clean 状态。

14.2.7 Ceph 容量耗尽解决方案

Ceph 存储集群通过 near full ratio 和 full ratio 来进行容量使用预警，通常当 Ceph 使用容量达到 near full ratio 设定的比率值时，Ceph 便会向管理员发出容量使用告警。此时管理员必须进行集群扩容操作，如果管理员未对 Ceph 集群的 near full ratio 预警做出正确响应，则可能出现两种导致 Ceph 集群不可用的情况：一种是在容量到达 near full ratio 后如果出现 OSD 故障，则很可能使 Ceph 容量骤升至 full ratio 值；另一种是在客户端 IO 高负载情况下大量数据写入使得 Ceph 容量迅速升至 full ratio 值。对 Ceph 而言，一旦容量到达 full ratio 值，则 Ceph 为防止数据丢失将停止一切数据访问。下面给出几个 Ceph 容量已达到 full ratio，且全部 Ceph 客户端已瘫痪的情况下，如何恢复 Ceph 数据访问的解决方案。

1. 存储扩容

容量耗尽最根本有效的解决方案就是扩容存储空间，即增加 OSD，这也是 Ceph 官方推荐的做法，通常在容量接近 near full ratio 时，便可准备扩容工作，扩容步骤可参考 14.2.2 节进行。扩容之后，Ceph 集群开始数据迁移，OSD 的 near full 比例便会持续下降，如下 OSD 在扩容之前 OSD near full 已达到 95% 以上，扩容后 OSD near full 值开始持续下降：

```
2017-01-11 14:28:14.588270 osd.26 10.170.5.63:6815/6592 425 : [WRN] OSD near full
(93%)
2017-01-11 14:28:22.741227 osd.31 10.170.5.64:6806/6892 415 : [WRN] OSD near full
(90%)
2017-01-11 14:28:27.957156 osd.10 10.170.5.62:6818/7883 385 : [WRN] OSD near full
(94%)
2017-01-11 14:28:19.866523 osd.24 10.170.5.63:6806/5424 418 : [WRN] OSD near full
(93%)
```

2. 紧急参数修改

存储扩容虽然是容量耗尽时最有效的方案，但是在 Ceph 容量达到 full ratio 之后才考虑扩容，如果此时身边没有多余的硬盘，或者正处于业务关键时期，增加 OSD 需要花费一定时间，此时可采取临时快速解决办法，即修改 mon_osd_full_ratio 和 mon_osd_nearfull_ratio 参数值，默认情况下这两个参数的值分别为 95% 和 85%，为了让 Ceph 可以临时响应客户端请求，可以适当调高 mon_osd_full_ratio 参数值，有两种方法可以实现：

(1) 修改 ceph.conf 配置文件

将新的 near full ratio 和 full ratio 参数值写入 Ceph 配置文件 ceph.conf 中, 如下:

```
[global]
mon osd full ratio = .98
mon osd nearfull ratio = .80
```

重新启动 Ceph 集群的 MON 进程, 使新的配置参数生效, 由于 full ratio 由之前的 95% 提升为现在的 98%, Ceph 将可以临时响应客户端请求, 此时除了通过命令行与 Ceph 集群交互, 进行不必要数据清除之外, 不要让虚拟机和块存储客户端往 Ceph 中写数据, 否则再次达到 98% 就真得没多少空间可往上调了。

(2) 动态注入新的配置参数

修改 Ceph 参数一个较为快速的方式就是通过 injectargs 命令进行动态修改, 例如要动态将 full ratio 参数值由 95% 调整为 98%, 可在运行时进行如下操作:

```
ceph mon tell \* injectargs '--mon-osd-full-ratio 0.98'
```

动态修改的参数将会即时生效, 修改完成后 Ceph 即可响应客户端请求, 但是此时千万不要让客户端往 Ceph 集群中写入数据, 作为一种临时性的解决办法, 可以将 Ceph 中某些不必要的数据清除, 如某些 Glance 镜像文件等, 还有就是设置某些非关键 POOL 的 pool size 值, 例如默认 pool size 为 3, 可以将 pool size 调为 2, min pool size 调为 1, 此时 Ceph 将进行数据重新平衡, 并且能释放出可观的可用容量, 待后续 Ceph 扩容完成后, 可再重新恢复 POOL 的 pool size 和 min pool size 值, POOL 的 size 和 min size 参数设置命令如下:

```
[root@controller1 ~]# ceph osd pool set images size 2
set pool 5 size to 2
[root@controller1 ~]# ceph osd pool set images min_size 1
set pool 5 min_size to 1
```

3. 直接文件删除

如果 Ceph 容量耗尽且不能访问时, 没有现存硬盘扩容, 临时性修改参数又不起作用, 可以直接删除 Ceph 集群中某些不必要的数据, 以 OpenStack 为例, 通过 OpenStack 的 Nova、Glance 和 Cinder 项目会采用 Ceph 作为存储后端, 其中 Nova 和 Cinder 的数据相对重要, 而 Glance 中的镜像或某些快照的清除并不会影响业务数据, 因此紧急情况下可以将其删除, 待扩容之后再重新上传或快照即可。Ceph 中的 Glance 镜像删除可按如下步骤操作:

(1) 查询 Nova 后端存储池中的镜像

OpenStack 的 Nova、Glance 和 Cinder 项目作为 Ceph 的客户端, 在 Ceph 集群中有分别对应的存储池, 如 vms、images 和 volume, 如下:

```
[root@controller1 ~]# ceph osd lspools
0 data,1 metadata,2 rbd,3 volumes,4 vms,5 images,
```

查看 vms 存储池中的虚拟机镜像, 如下:

```
[root@controller1 ~]# rbd ls vms
f0717c2e-3765-4964-9244-be235925892e_disk
```

查询 vms 中虚拟机镜像的 parent，如下：

```
[root@controller1 ~]# rbd info vms/f0717c2e-3765-4964-9244-be235925892e_disk
rbd image 'f0717c2e-3765-4964-9244-be235925892e_disk':
.....
parent: images/8ca07a37-01c5-4472-be3b-91e0ad0e5c2e
.....
```

(2) 核对 Glance 后端存储池镜像

vms 存储池中虚拟机镜像的 parent 信息表明了该虚拟机所使用的镜像文件，这些镜像文件存储在 image 存储池中。反之，如果 image 中的镜像没有在 vms 中出现，则可将其删除，如下：

```
[root@controller1 images]# rbd ls images
8ca07a37-01c5-4472-be3b-91e0ad0e5c2e
f6324b23-4249-4067-a885-b44cde7b758a
```

因镜像 f6324b23-4249-4067-a885-b44cde7b758a 未被使用，因此可将其删除。

(3) 删除未使用镜像

查询要删除镜像的 prefix，如下：

```
[root@controller1 images]# rbd info images/f6324b23-4249-4067-a885-b44cde7b758a
rbd image 'f6324b23-4249-4067-a885-b44cde7b758a':
size 40162 kB in 5 objects
order 23 (8192 kB objects)
block_name_prefix: rbd_data.10464d305d48
format: 2
features: layering, striping
stripe unit: 4096 kB
stripe count: 1
```

进入 OSD 容量块耗尽的存储节点，并进入 OSD 进程的 current 目录，假设 osd.1 空间即将耗尽，则进入 osd.1 的 current 目录，如下：

```
[root@compute2 ~]# cd /var/lib/ceph/osd/ceph-1/current
```

搜索与要删除镜像相关的文件对象，如下：

```
find . -type f -name *10464d305d48* > ./delete_objects.txt
```

删除 delete_objects.txt 中的全部文件，如下：

```
cat delete_objects.txt | while read line
do
rm -rf ${line}
done
```

文件删除之后，注意观察 Ceph 存储空间是否有所变化。通常而言，如果 image 存储池中存储了很多大容量镜像，则此方式可以释放较多的空间。不过，对于 TB 或 PB 级别的 Ceph 存储池，此方法释放的空间相对整体容量而言实在太小，建议采用前面两种方式。

14.2.8 Ceph 常用命令使用参考

1. 集群操作命令

(1) 启动 OSD 进程

```
[root@computer1 ~]# service ceph start osd.0
```

(2) 停止 OSD 进程

```
[root@computer1 ~]# service ceph stop osd.0
```

(3) 启动 MON 进程

```
[root@computer1 ~]# service ceph start mon.computer1
```

(4) 停止 MON 进程

```
[root@computer1 ~]# service ceph stop mon.computer1
```

(5) 查看 Ceph 存储空间

```
[root@controller1 ~]# ceph df
```

(6) 删除节点上全部 Ceph 数据包

```
[root@controller1 ~]#ceph-deploy purge computer1
```

```
[root@controller1 ~]#ceph-deploy purgedata computer1
```

(7) 为 Ceph 创建 admin 用户同时为其创建秘钥

```
[root@controller1 ~]#ceph auth get-or-create client.admin mds 'allow' osd 'allow *' \
mon 'allow *' > /etc/ceph/ceph.client.admin.keyring
```

(8) 查看 Ceph 授权用户

```
[root@controller1 ~]# ceph auth list
```

(9) 删除 OSD 认证信息

```
[root@controller1 ~]# ceph auth del osd.0
```

(10) 查看集群配置信息

```
[root@computer1 ~]# ceph daemon mon.computer1 config show | more
{
  "name": "mon.computer1",
  "cluster": "ceph",
  "debug_none": "0/5",
  .....
}
```

(11) 查看 OSD 的 admin socket

```
[root@computer1 ~]# ceph-conf --name osd.0 --show-config-value admin_socket
/var/run/ceph/ceph-osd.0.asok
```

(12) 查看 OSD 进程日志位置

```
[root@compute1 ~]# ceph-conf --name osd.1 --show-config-value log_file
/var/log/ceph/ceph-osd.1.log
```

(13) 查看 OSD 进程配置参数

```
[root@compute1 ~]# ceph --admin-daemon /var/run/ceph/ceph-osd.0.asok config show |more
{ "name": "osd.0",
  "cluster": "ceph",
  "debug_none": "0\5",
  "debug_lockdep": "0\1",
```

(14) 动态配置集群参数

```
[root@controller1 ~]# ceph mon tell \* injectargs '--mon-osd-full-ratio 0.98'
mon.compute1: injectargs:mon_osd_full_ratio = '0.98'
mon.compute2: injectargs:mon_osd_full_ratio = '0.98'
mon.storage1: injectargs:mon_osd_full_ratio = '0.98'
```

(15) 删除 OSD

```
[root@controller1 ~]# ceph osd rm 0
removed osd.0
```

(16) 将 OSD 从 CRUSH Map 中删除

```
[root@controller1 ~]# ceph osd crush rm osd.0
```

(17) 设置 OSD 权重

```
[root@controller1 ~]# ceph osd crush reweight osd.0 1.0
reweighted item id 0 name 'osd.0' to 1 in crush map
```

(18) 暂停 Ceph 集群

```
[root@controller1 ~]# ceph osd pause
set pauserd,pauserw
```

(19) 重启 Ceph 集群

```
[root@controller1 ~]# ceph osd unpause
unset pauserd,pauserw
```

(20) 查看一个 PG 的 MAP 信息

```
[root@controller1 ~]# ceph pg map 3.f
osdmap e138 pg 3.f (3.f) -> up [0,2,4] acting [0,2,4] //osd.0为Primary OSD, 其他
两个为relicas
```

(21) 查看一个 PG 的详细信息

```
[root@controller1 ~]# ceph pg 3.3 query
```

2. 集群状态监控命令

(1) 查看集群健康情况

```
[root@controller1 ~]# ceph health
HEALTH_OK
```

(2) 查看集群运行状态

```
root@controller1 ~]# ceph status
```

(3) 查看 OSD 状态

```
[root@controller1 ~]# ceph osd stat
osdmap e119: 5 osds: 5 up, 5 in
```

(4) 查看 OSD Tree

```
[root@controller1 ~]# ceph osd tree
```

(5) 查看 OSD dump 信息

```
[root@controller1 ~]# ceph osd dump
```

(6) 查看 MON dump 信息

```
[root@controller1 ~]# ceph mon dump
```

(7) 查看监控节点 quorum 状态

```
[root@controller1 ~]# ceph quorum_status
```

(8) 查看 PG 状态

```
[root@controller1 ~]# ceph pg stat
v1606: 576 pgs: 360 active+remapped, 216 active+clean; 147 MB data, 606 MB used,
45418 MB / 46025 MB avail; 4/126 objects degraded (3.175%)
```

(9) 查看 PG 中 stuck 状态

```
[root@controller1 ~]# ceph pg dump_stuck unclean
ok
[root@controller1 ~]# ceph pg dump_stuck inactive
ok
[root@controller1 ~]# ceph pg dump_stuck stale
ok
```

3. 存储池操作命令

(1) 查看集群中的 POOL

```
[root@controller1 ~]# ceph osd lspools
0 data,1 metadata,2 rbd,3 volumes,4 vms,5 images
```

(2) 创建一个 POOL

```
[root@controller1 ~]# ceph osd pool create warrior 256
pool 'warrior' created
```

(3) 删除一个 POOL

```
[root@controller1 ~]# ceph osd pool delete warrior warrior --yes-i-really-
really-mean-it
pool 'warrior' removed
```

(4) 显示 POOL 详细信息

```
[root@controller1 ~]# rados df
```

(5) 为 POOL 创建快照

```
[root@controller1 ~]# ceph osd pool mksnap vms vms-snap
created pool vms snap vms-snap
```

(6) 删除 POOL 快照

```
[root@controller1 ~]# ceph osd pool rmsnap vms vms-snap
removed pool vms snap vms-snap
```

(7) 查看 POOL 的副本数

```
[root@controller1 ~]# ceph osd pool get vms size
size: 3
```

(8) 查看 POOL 的最小副本数

```
[root@controller1 ~]# ceph osd pool get vms min_size
min_size: 2
```

(9) 查看 POOL 的 PG 数目

```
[root@controller1 ~]# ceph osd pool get vms pg_num
pg_num: 128
```

(10) 设置 POOL 的 PG 数量

```
[root@controller1 ~]# ceph osd pool set vms pg_num 256
set pool 4 pg_num to 256
```

(11) 设置 POOL 的 Rule

```
ceph osd pool set SSD crush_ruleset 1 //POOL名称为SSD, Rule ID为1
```

4. RBD 操作命令

(1) 查看 POOL 里的所有镜像

```
[root@controller1 ~]# rbd ls vms
f0717c2e-3765-4964-9244-be235925892e_disk
[root@controller1 ~]# rbd ls images
8ca07a37-01c5-4472-be3b-91e0ad0e5c2e
f6324b23-4249-4067-a885-b44cde7b758a
[root@controller1 ~]# rbd ls volumes
volume-27b33034-950f-415c-98bd-9c462367efdb
volume-8c6dc44a-6ec3-4187-8de9-5daa1dec0b01
volume-c085140d-0f8f-43e7-a77d-711a80293732
```


(2) 查看 POOL 中镜像信息

```
[root@controller1 ~]# rbd info volumes/volume-27b33034-950f-415c-98bd-9c462367efdb
```

(3) 在存储池中创建镜像

```
[root@controller1 ~]# rbd create -p images --size 1024 warrior //创建1024MB的镜像
[root@controller1 ~]# rbd ls images
warrior
8ca07a37-01c5-4472-be3b-91e0ad0e5c2e
f6324b23-4249-4067-a885-b44cde7b758a
```

(4) 调整镜像大小

```
[root@controller1 ~]# rbd resize -p images --size 2048 warrior
Resizing image: 100% complete...done.
```

(5) 删除存储池中的镜像

```
[root@controller1 ~]# rbd rm -p images warrior
Removing image: 100% complete...done.
```

(6) 为镜像创建一个快照

```
[root@controller1 ~]# rbd snap create images/warrior@warrior-snap
//语法为: rbd snap create pool_name/image_name@snap_name
```

(7) 查看镜像快照

```
[root@controller1 ~]# rbd snap ls -p images warrior
```

```
SNAPID NAME SIZE
```

```
4 warrior-snap 2048 MB
```

```
[root@controller1 ~]# rbd info images/warrior@warrior-snap
```

(8) 删除镜像文件快照

```
root@controller1 ~]# rbd snap rm images/warrior@warrior-snap
```

(9) 删除某个镜像的全部快照

```
[root@controller1 ~]# rbd snap purge -p images warrior
Removing all snapshots: 100% complete...done.
```

(10) 导出 Glance 后端 POOL 中的镜像

```
[root@controller1 ~]# rbd export -p images --image 8ca07a37-01c5-4472-be3b-91e0ad0e5c2e \
cirros.img
Exporting image: 100% complete...done.
[root@controller1 ~]# ls -l cirros.img
-rw-r--r-- 1 root root 13287936 Feb 5 09:24 cirros.img
```

(11) 导出 Cinder 后端 POOL 中的云硬盘数据

```
[root@controller1 ~]# rbd export -p volumes --image \
```

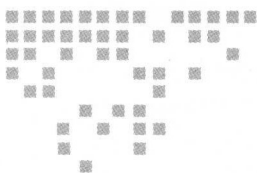
```
volume-27b33034-950f-415c-98bd-9c462367efdb cinder-volume.20170207
Exporting image: 100% complete...done.
[root@controller1 ~]# ls -l cinder-volume.20170207
-rw-r--r-- 1 root root 2147483648 Feb  5 09:27 cinder-volume.20170207
```

14.3 本章小结

Ceph 存储集群是目前极为热门的开源存储技术，随着云计算技术，尤其是 OpenStack 在企业用户中的不断落地实施，Ceph 作为一种具备高可用、高扩展、高性能以及数据自我修复的统一存储，越来越多的用户正在使用 Ceph 构建大规模的海量存储池。本章从 Ceph 硬件选型、配置参数使用等方面介绍了如何进行 Ceph 部署的前期规划，对于 Ceph 集群，SSD 固态硬盘已成为标配，本章介绍了常见的 SSD 使用场景，并介绍了如何通过修改 Ceph 的 CRUSH Map 信息来自定义使用 Ceph 集群。为了提高 Ceph 集群的运行性能，本章还从硬件和软件角度阐述了对 Ceph 进行性能调优的方法，由于 Ceph 参数众多，因此 Ceph 集群的性能调优也将会是漫长的过程。针对 Ceph 存储在日常运行维护中经常遇到的情况，如更换 OSD 硬盘、集群扩容、服务器维护、监控节点维护、日志维护以及 OSD 存储空间耗尽等，本章均给出了具体的解决参考方案。通过对本章的学习，读者可以深入了解 Ceph 集群规划、Ceph 集群调优以及 Ceph 集群维护等内容。

扩展篇

■ 第15章 Docker 容器部署 OpenStack



Docker 容器部署 OpenStack

Docker 是目前容器领域极为热门的技术，Docker 的出现使得容器技术再次吸引了整个 IT 世界的关注，红极一时的 Docker 曾被认为是 OpenStack 的宿敌，并将最终战胜 OpenStack 成为 IT 技术革命的引领者。然而，事实并非如此，OpenStack 以 Big Tent 模式将自身当成一个强大的集成引擎，将 Docker 容器技术与现有的 OpenStack 项目不断集成，并以 Docker 为基础以全新的项目使命孵化了诸多整合容器技术的 OpenStack 项目，从目前 OpenStack 和 Docker 生态圈的发展情况来看，二者非但没有走向对立，反而彼此互补、相得益彰并且都发展得欣欣向荣。本章将介绍 OpenStack 与 Docker 互补融合之后的 OpenStack 社区项目之一，也是最近非常热门的 OpenStack 容器化部署技术和微服务架构的践行方案，即 OpenStack 的 Docker 容器部署项目 Kolla。Kolla 的出现，对于 OpenStack 社区的代码及功能交付带来了革命性的影响，对于被诟病已久的 OpenStack 复杂多样、难以部署和难以升级维护等弊端带来了彻底的解决方案。本章将从 Kolla 项目的起源、项目使命，以及 Kolla 的现状和内部功能组件方面对其进行详细介绍，并通过 Kolla 实战部署的形式来讲解如何利用 Kolla 简单优雅地部署 OpenStack。

15.1 OpenStack 与 Docker

15.1.1 容器与虚拟机的现状

近两年来，随着 Docker 的出现，问世已久的容器（Container）技术再次成为 IT 领域的热门方向，围绕着 Docker 生态圈，出现了各种容器调度编排和管理技术，如 Docker Swarm、Mesos 和 Kubernetes（k8s）等，现今，Docker 似乎已成为容器的代名词。此外，随

随着容器技术的成熟，应用微服务的概念也被不断提及和推广，从社区的关注度、生态圈的发展和现有 IT 架构及应用的影响来看，尽管容器技术目前还未普遍应用于大规模生产系统中，但是仍然可以认为以 Docker 为代表的容器技术是继 OpenStack 之后又一革命性的社区开源技术。伴随 Docker 的来势汹汹，正是“当红之年”的 OpenStack 从 Docker 诞生起便被放在了其对立面进行对比，关于二者孰优孰劣的争论也一直伴随着 Docker 及其生态圈的成长进化，直到 OpenStack 的包容发展对 Docker 进行整合共存后才稍见消停。本节中，我们并未过多地介绍容器技术的发展历史以及 Docker 的技术原理细节，而是着重从容器和虚拟机在实际应用中的角色来客观介绍二者的异同，以期读者对 Docker 容器和 OpenStack 云计算有客观的认识。

首先，容器不是虚拟机，虽然在某些方面二者具有相似性，但更多的是存在的差异，因此不应该将容器与虚拟机放在对等的位置上进行轻重、快慢和安全与否等绝对性的比较。容器是系统中一个隔离、可移植的环境，该环境中包含了运行应用程序所需的依赖和库，因而用户可以在其中运行相应的应用程序，容器与虚拟机的相同之处在于，多个容器之间可以共享系统的计算、存储和网络资源，不同之处在于容器对宿主机资源的共享无须 Hypervisor 的介入，不同于虚拟机通过 Hypervisor 在虚拟机操作系统之间调度底层物理资源，在容器技术中，宿主机上的所有容器共享同一操作系统内核，并通过命名空间 (Namespace) 和资源组控制技术 (CGroup) 实现不同容器的隔离和资源共享，或者说容器采用的是进程级别的隔离，而虚拟机采用的是操作系统级别的隔离，容器与虚拟机之间的差异如图 15-1 所示。

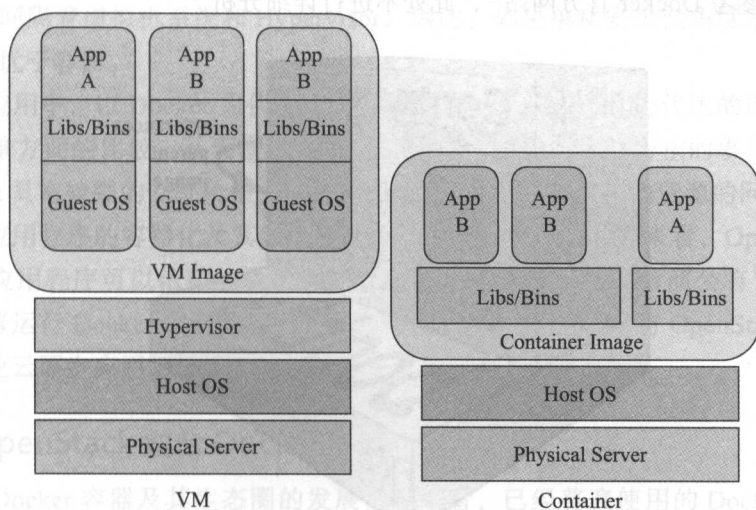


图 15-1 虚拟机与容器

除了在资源隔离与共享方面的差异，OpenStack 虚拟机与 Docker 容器在应用领域上也不同。OpenStack 虚拟机属于 IaaS 层面，提供的是计算、网络 and 存储等基础架构设施资源，

Docker 作为一个应用进程可以运行在虚拟机上，即以 OpenStack 为代表的虚拟机主要以资源为中心，而 Docker 容器属于 PaaS 层面（容器也被称为 CaaS），主要以应用为中心，其解决的主要问题是应用程序的依赖封装、隔离和移植，以及对复杂完整的应用程序进行拆分和容器化，而这些拆分后的进程通常称为微服务（MicroService）。尽管微服务架构的概念早已存在，但是容器技术的成熟使用才是微服务架构实现的关键，拆分的微服务被容器化之后，通过容器编排管理引擎使所有的容器相互通信并组成功能完整的应用程序，同时由容器编排引擎保证容器在节点之间的调度和微服务的高可用性，是目前容器技术背景下应用程序服务架构的主要发展方向，也是 Docker 容器技术在应用程序领域的主要应用。

与虚拟机相比，容器技术尤其是 Docker 容器能够快速普及的主要原因之一，是 Docker 特有的镜像管理机制。容器技术很早便有，Docker 成为容器的代表，并不在于 Docker 发明了容器，而在于其创新性的容器镜像管理机制，需要指出的是，早期的 Docker 基于开源 LXC 容器管理引擎，而目前改为基于 Go 语言实现的 Libcontainer。Docker 镜像采用层级继承方式，不同的子镜像可以共享同一个底层父镜像，如图 15-2 所示。在图 15-2 中，最下层的 Debian 发行版本 Linux 系统称为 Base 镜像（也称为第一层镜像），除了 Base 镜像之外的其他层镜像称为 Parent 镜像，例如当用户在 Dockerfile 文件中添加了安装部署 emacs 软件的配置后便得到一层新的镜像，这里将其称为第二层镜像，之后用户再通过 Dockerfile 文件在第二层镜像的基础上添加了安装部署 Apache 的配置后，又得到了第三层镜像，由于继承关系，第三层镜像中同时部署了 emacs 和 Apache 软件，当用户通过 Docker 命令启动第三层镜像时，会在其上增加一个可写层，这个可写层便是容器。关于 Docker 容器与容器镜像之间的关系可参考 Docker 官方网站^①，此处不进行详细分析。

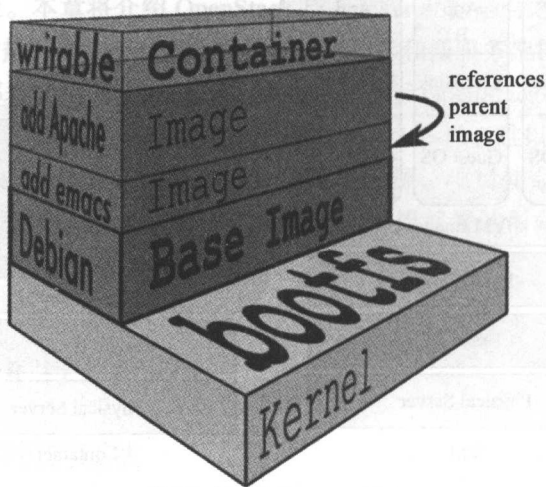


图 15-2 Docker 容器镜像的分层实现

① <https://www.docker.com/>

总体而言，容器与虚拟机之间的异同主要有以下几个方面：

- 从隔离层面来看，虚拟机实例是一个完整的操作系统，虚拟机之间通过 Hypervisor 进行隔离并共享宿主机资源，而容器实例作为系统进程存在于操作系统中，并与宿主机共享操作系统内核，通过内核提供的命名空间、CGroup 等运行时隔离功能为容器内部服务提供独立的用户域、文件系统、网络以及进程运行空间。因此，从隔离程度而言，虚拟机是一种系统级别的完全隔离，而容器为进程级别的隔离。
- 从镜像层面来看，容器更多应用于 PaaS 层面的特定服务，因而其构建的镜像通常只包含运行该服务所需的依赖和库文件，很多常见服务的容器镜像仅有几十或几 MB。相比而言，虚拟机则提供了包括内核在内的通用进程运行环境，而非仅包含特定服务的运行环境，因而虚拟机镜像通常体积臃肿但是功能集齐全，虚拟机镜像一般在几百 MB 或几 GB。从镜像体积上来看，容器“更轻”，而虚拟机“较重”，因此如果单从启动速度、运行性能和系统资源消耗方面来对比，则容器要优于虚拟机。
- 从使用方式来看，容器的使用方式更多是倾向于开箱即用，即容器镜像提供的是一个封装完成、准备就绪、不可更改并可立即启用的服务资源。而虚拟机则倾向于将使用配置权交给用户，让用户根据自身需求，对系统进行自定义初始化，并使用类似 Ansible、Chef 和 Puppet 等自动化配置工具进行基础设施的配置管理。
- 从安全层面来看，虚拟机实例之间是一种物理机制的隔离，位于不同虚拟机上的进程之间隔着虚拟机系统和 Hypervisor，因此，如果单从安全性角度来考虑，则虚拟机要优于容器。

在实际应用中，以 Docker 为代表的容器技术和以 OpenStack 为代表的虚拟机技术不应局限于某些单方面的比较，从而便得出谁优谁劣的结论。到目前为止的事实证明，Docker 与 OpenStack 具有极强的互补价值，OpenStack 着重解决了基础架构资源的问题，而 Docker 的重点在于应用程序的容器化及其编排和微服务领域，从这一层面来看，OpenStack 提供了容器平台和应用程序可以依赖的资源和服务，就目前而言，在运营商公有云或 OpenStack 私有云上部署运行 Docker 容器已成为一种趋势，Docker 容器技术与 OpenStack 虚拟机的互补结合为企业云原生应用的部署实施提供了完善的解决方案。

15.1.2 OpenStack 融合 Docker

从目前 Docker 容器及其生态圈的发展情况来看，已经普遍使用的 Docker 容器并未如之前争论的一样走到虚拟机的对立面，Docker 容器及其生态圈发展欣欣向荣，而虚拟机也并未如曾经的预言走向死亡，相反容器与虚拟机目前一直和谐共处、相得益彰，并且还出现了很多打破二者壁垒，并将容器的便捷性和虚拟机的安全性相互整合的开源项目，如“容

器式操作系统” RancherOS^①、“容器式虚拟机” Hyper^②以及“虚拟机式容器” LXD^③。而作为受 Docker 冲击最强的 OpenStack，其在“大帐篷”（Big Tent）模式下也不断对 Docker 进行支持和融合，到目前为止，OpenStack 中对 Docker 容器进行整合兼容的项目均是 OpenStack 社区参与度较为热门的项目，如从早期的 Nova Docker Driver 和 Heat Docker Driver 到现在仍然很活跃的 Magnum、Murano、kuryr 和 Kolla 等项目，都是 OpenStack 社区拥抱 Docker 的体现。下面对 OpenStack 社区与 Docker 的结合项目进行简要介绍。

（1）Nova Docker Drive

Nova Docker Drive 项目是 OpenStack 社区与 Docker 的第一次集成，该项目的设计思想就是将容器当作虚拟机来处理，同时将 Docker 引擎当成一种新的 Hypervisor，即类似于为 Nova Compute 提供了一个新的 Docker 驱动。Nova 与 Docker 集成较为简单，OpenStack 只需调用 Docker RESTful API 即可操作容器。Nova Docker Drive 项目有很多优点，比如通过 Nova-scheduler 来调度资源、通过 Nova 虚拟机的 API 来操作容器以及使用 Neutron 来管理 Docker 容器网络等，但是容器与 Nova 虚拟机毕竟存在很多差异，因此通过与 Nova 的集成并不能很好地实现 Docker 容器的很多高级功能，如容器关联、端口映射以及 Docker 的各种网络模式。

（2）Heat Docker Driver

Heat Docker Driver 项目可以认为是 OpenStack 社区对 Nova Docker Driver 项目的补充，由于 Nova Docker Driver 项目不能实现很多 Docker 高级功能，因此社区希望通过 Heat Docker Driver 项目来解决这个问题。该项目的设计思路是通过在 Heat 中新增一个 Resource 插件来与 Docker 集成，Heat Docker Driver 的新增插件通过 REST API 直接与 Docker 交互，同时不需要与 Cinder、Nova 和 Neutron 等项目交互，由于 Heat Docker Driver 与 Docker API 完全兼容，因此可以在 OpenStack 中实现 Docker 的很多高级功能，不过由于该项目仅通过 Heat 的 API 来与 Docker 交互，因此没法对 Docker 进行资源调度，用户必须通过模板形式指定需要部署服务的目标容器，所以并不适合大规模的应用部署场景。

（3）Magnum

OpenStack 社区最初希望从 Nova 和 Heat 两个项目上来对 Docker 进行整合，但是发现整合效果难以令人满意，因此便决定另起一个全新的项目来与 Docker 整合，于是便创建了 Magnum 项目。Magnum 项目创建于 2014 年的巴黎峰会，其主要目的是提供容器服务，同时与 K8S、Mesos、Swarm 和 CoreOS 等容器管理系统集成。Magnum 的两个主要服务分别是 Magnum API 和 Magnum Conductor，Magnum API 主要接收客户端请求，并将请求发送给 Magnum 后端，而 Magnum API 支持的后端包括 K8S、Swarm、CoreOS 和 Docker 等。此

① <https://github.com/rancher/os/>

② <https://github.com/hyperhq/hyperd>

③ <https://github.com/lxc/lxd>

外, Magnum Conductor 主要用于客户端请求到后端的转发,即 Magnum Conductor 通过解析客户端请求,之后再将其转发至合适的 Magnum 后端中。Magnum 项目通过集成 K8S、Swarm 等容器管理系统,使得用户在 OpenStack 集群中不仅可以提供 IaaS 服务,还可以通过 K8S、Swarm 来提供容器服务。

(4) Murano

Murano 最初是 OpenStack 上提供应用服务目录的项目, Murano 与 K8S 集成后, K8S 成为了 Murano 的应用服务, 此处将其列为 OpenStack 与 Docker 容器的整合项目, 主要也是因其整合 K8S 后, 用户可以通过 Murano 部署 Pod、Service、Replication 和 Controller 等服务。用户在通过 Murano 选中所需的应用服务之后, OpenStack 便通过 Heat 来部署选中的应用服务。

(5) Kuryr

Kuryr 项目的目的在于将 OpenStack 中的 Neutron 网络与 Docker 网络整合, 使得 Docker 可以使用 Neutron 网络。在 Neutron 与 Docker 的网络整合中, Kuryr 并未扮演网络解决方案的角色, 而是作为 Neutron 和 Docker 网络之间的“整合桥梁”, 以便将 Neutron 网络及其服务传递至 Docker 中。

(6) Kolla

在 OpenStack 与 Docker 的整合过程中, 前述几个项目都是为了能够更好地在 OpenStack 环境中使用容器或提供容器服务, 而 Kolla 项目的目标却是将 OpenStack 服务项目容器化, 并通过容器化的方式来部署 OpenStack 集群。在 OpenStack 的部署及运维过程中, 存在各种部署方式, 如 Tripe-O、RDO、Fuel、Puppet、Chef 和 Ansible 等, 但是这些部署方式都要求用户对 OpenStack 具有较深的理解, 并需要配置繁杂的参数, 即虽然使用 Puppet、Chef 和 Ansible 等工具可以实现自动化的大规模部署, 但是却要求用户重新去学习并掌握一个复杂的自动化运维工具, 这不仅没有简化 OpenStack 的部署, 反而增加了用户对新工具的学习成本和部署难度。此外, 即使花费大量时间掌握了上述部署工具, 并能够通过这些工具成功部署 OpenStack, 但是 OpenStack 的升级更新又是一个头疼的问题。而 Kolla 项目的出现便是为了解决 OpenStack 的部署和升级更新的难题, Kolla 通过 Docker 镜像将每个 OpenStack 服务镜像化, 并通过容器形式来部署运行 OpenStack 服务, 由于 Docker 镜像的原子性, 当需要升级更新 OpenStack 服务时, 只需停止老版本镜像 Docker 容器, 并启动新版本镜像 Docker 容器即可, 而如果需要回退, 则仅需删除新版本镜像容器, 并重新启动老版本镜像容器即可。

随着 Docker 的不断发展, OpenStack 社区与 Docker 集合的项目也在不断增加, 而在这些与容器集成的项目中, 用户应该如何选择最佳的容器服务, 这需要集合用户自身的情况来决策。例如, 用户需要将以前运行在 VM 上的应用负载迁移至 Docker 容器上, 则可以使用 Nova Docker Driver 项目来实现, 而如果用户希望使用 Docker 的某些高级功能部署一个小规模的容器集群, 则 Heat Docker Driver 是最佳选择, 如果用户希望通过 OpenStack

集成诸如 K8S、Swarm 等容器管理工具对大规模 Docker 集群进行管理, 则 Magnum 是不二选择, 此外通过 Murano 集成的 K8S 也可管理 Docker 容器, 只是 Murano 聚焦的功能在于应用目录服务, 而 Magnum 则专注于容器服务。最后, 对于很多 OpenStack 用户而言, 最迫切需要的应该是 Kolla 项目, 不论使用哪一种 OpenStack 与容器集合的项目, 如果不能成功部署和运维 OpenStack 集群, 则任何基于 OpenStack 基础架构资源的服务都无法使用。Kolla 项目的意义, 在于通过 Docker 容器化方法部署 OpenStack 高可用集群, 并通过 Docker 镜像以原子方式升级回退 OpenStack。通过 Kolla 项目, 用户可以直接在 Kolla 项目上游维护的镜像仓库中下载 OpenStack 各个服务的 Docker 镜像, 并直接通过 Kolla-ansible 对其进行一键部署, 同时 Kolla 也提供了用户自定义容器镜像的方法, 使得具有特殊需求的用户可以自定义 Docker 容器, 并进行满足自身需求的 OpenStack 部署或升级。

因此, 作为一个运维管理员, Kolla 项目才是 OpenStack 与 Docker 容器集成中最有价值和应该优先考虑的项目。而本章主要介绍的也正是通过 Kolla 项目以 Docker 容器化方式部署 OpenStack, 通过 OpenStack 社区的 Kolla 项目, 运维管理人员以往在部署实施和版本升级过程中的很多问题都将迎刃而解, 本文的后续章节将对 Kolla 项目进行更为详细的介绍, 并通过 Kolla 进行 OpenStack 的部署实践。

15.2 Kolla 项目介绍

15.2.1 Kolla 项目使命

OpenStack 从出现至今, 其复杂的组件和繁多的配置选项就一直被诟病, 尤其对于很多初次接触 OpenStack 的用户而言, 部署和管理 OpenStack 就是一件令人恐惧的事情, 更不用提要跟上社区每年两个版本的更新速度, 而即使有多年 OpenStack 运维经验的工程师也不敢轻易对其升级。也正因如此, OpenStack 社区才出现了很多像 Tripe-O 和 Ironic 一样专注于 OpenStack 部署的全新项目, 同时也出现了很多集合第三方自动化部署软件的 OpenStack 子项目, 如 Openstack-Chef、Openstack-Puppet 和 Openstack-ansible 等, 然而这些项目要么从另外一个维度变相增加了 OpenStack 部署的难度, 要么没有从根本上解决 OpenStack 的运维难度, 如版本的升级和回退等问题。

从架构设计的高层次来看, OpenStack 的服务组件架构设计是非常优雅的, 如图 15-3 所示, OpenStack 内部服务组件彼此之间具有原子独立性, 相互之间通过各自的 API 进行互访, 同时各个服务组件也可独立运行, 即服务组件之间是一种多内聚、轻耦合的通用设计思想, 因此从顶层设计来看, OpenStack 内部架构显得非常优雅。

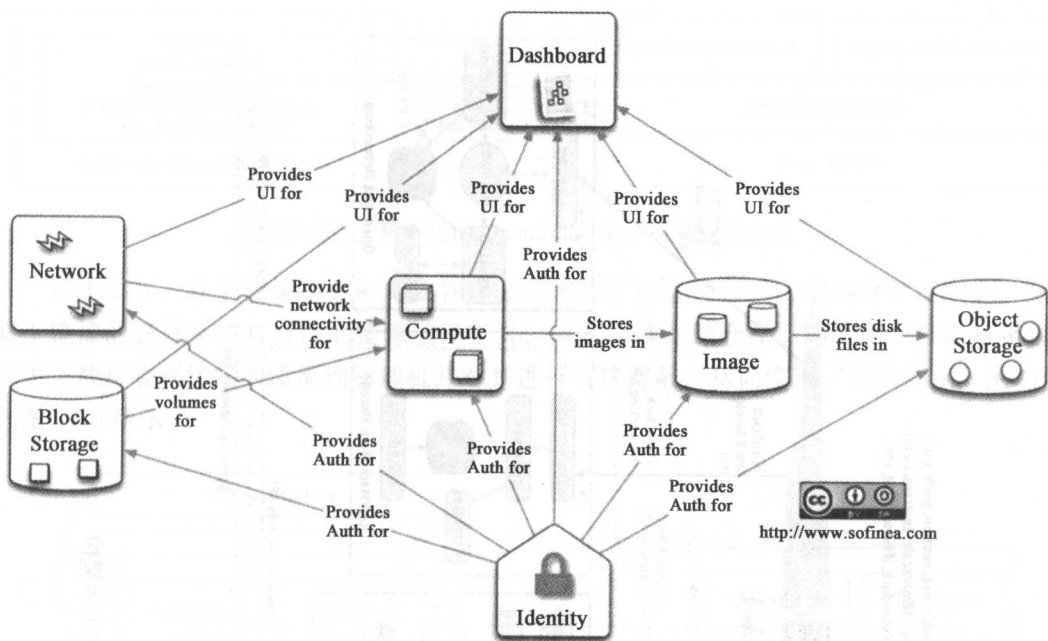


图 15-3 OpenStack 内部组件抽象架构

但是，在实际的实现和使用过程中，OpenStack 并非如图 15-3 一样显得“清纯可人”，其项目内部存在诸多插件且彼此之间存在错综复杂的交互，如图 15-4 所示。对于任何一个 OpenStack 新手，尤其是运维工程师而言，图 15-4 都足以起到震慑作用，这还不包括“大帐篷”中的其他几十个项目。

面对 OpenStack 如此众多的服务组件，以及各个组件内部繁杂多样的插件驱动和不断更新迭代的 OpenStack 版本，用户如何才能优雅地使用 OpenStack 并对其进行生命周期管理，从而将自己的业务系统安全平稳地整合到 OpenStack 云平台上，一直都是 OpenStack 社区在考虑并需要解决的问题，而 Docker 容器的出现，正好为 OpenStack 社区解决这一问题提供了极佳的思路。

就目前的 OpenStack 生命周期管理方式而言，可以分为两类，即基于包的管理方式和基于镜像的管理方式^①。基于包的管理方式通常需要借助 Puppet、Chef 和 Ansible 等配置管理工具，但是由于 OpenStack 每个服务都被部署在不同的物理服务器或者不同的操作系统上，往往使得这一类型的生命周期管理工具效率极低，此外由于 OpenStack 服务的复杂性和部署的灵活性，基于包的生命周期管理工具也很难保持和提供不同用户不同阶段的部署需求，再者基于包的管理方式通常需要在网络中来回传输软件包，如图 15-5 所示，当集群规模较大时，这种管理方式在速度上难以让人接受。

① <https://allthingsopen.com/2014/10/22/a-demonstration-of-kolla-docker-and-kubernetes-based-deployment-of-openstack-services-on-atomic/>

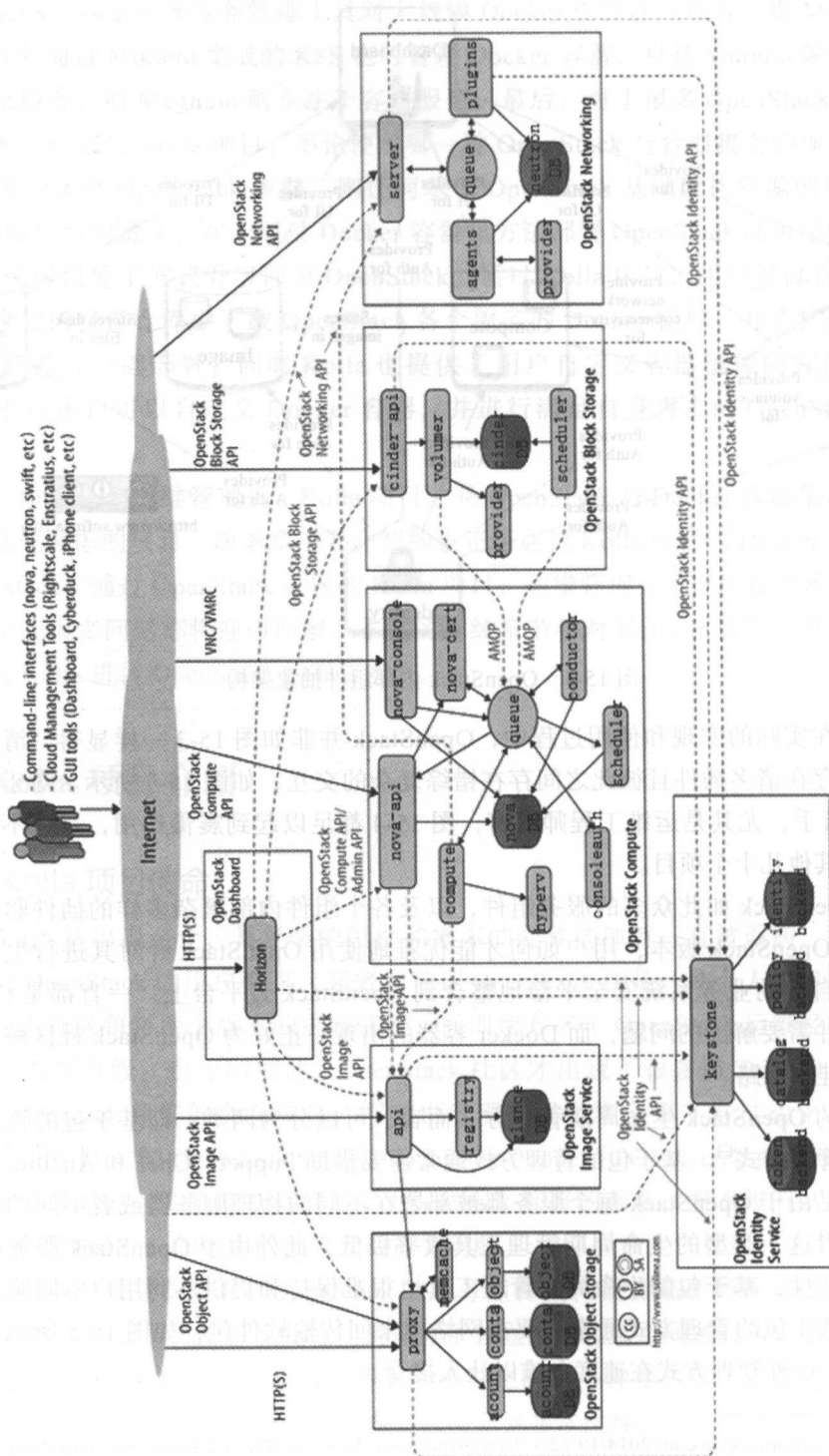


图 15-4 OpenStack 内部组件实际架构

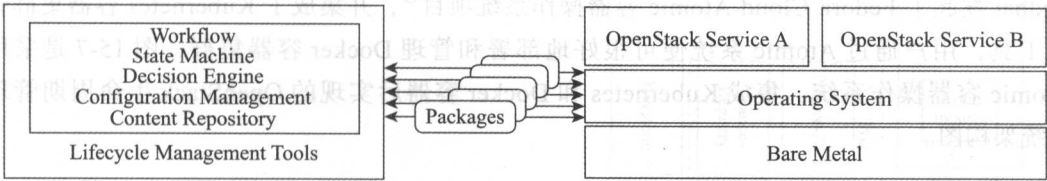


图 15-5 基于包的 OpenStack 生命周期管理方式

相比而言，基于镜像的生命周期管理方式解决了包管理方式中的各种性能问题，通常基于镜像的管理系统有自己的镜像编译工具，以及镜像存储仓库，同时用户可以从镜像仓库中下载封装制作好的镜像到物理机并对其进行直接部署，从而实现“开箱即用”的功能，如图 15-6 所示。

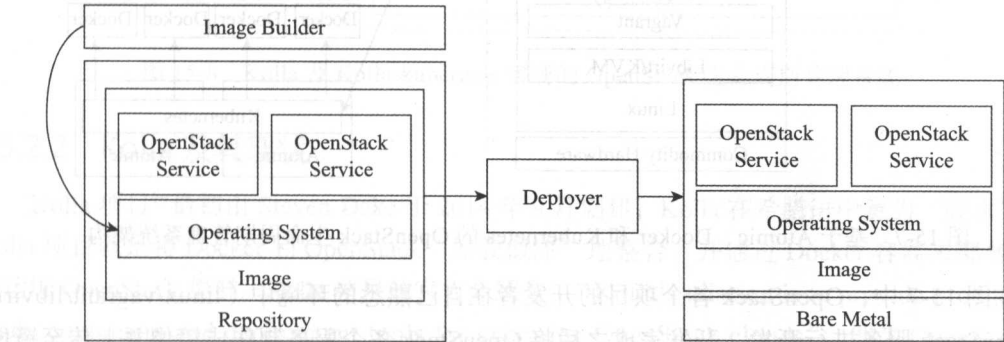


图 15-6 基于镜像管理系统的典型应用方式

但是，基于镜像的管理系统也存在一些问题，例如，由于镜像尺寸通常较大，而某个服务极小的配置变更就要重新编译并推送一个大尺寸的镜像，这种方式未免显得大动干戈。更重要的是，基于镜像的管理方式并未解决 OpenStack 服务之间复杂的依赖性问题，仅是将此问题提前放到镜像编译过程中而非运行时才解决。对于很多运维管理工程师而言，一个理想的 OpenStack 生命周期管理工具应该具备以下几点：

- ❑ 服务隔离，同时具有便携、轻量和可移植特性。
- ❑ 服务之间的运行时依赖关系易于描述定义。
- ❑ 服务易于升级更新，运行无须复杂设置。
- ❑ OpenStack 以外的生命周期管理服务。

Docker 容器以及容器编排调度引擎 Kubernetes 的集合使用便能实现上述 OpenStack 生命周期管理系统的要求。Docker 为服务提供了很好的隔离机制，同时 Docker 实现的镜像管理机制，使得开发人员可以发布便携的容器镜像至公有或私有的镜像仓库中，而运维人员从镜像仓库下载镜像并将其部署在不同的平台上。为了更好地支持 Docker 容器技术，

Redhat 发起了 Fedora Cloud Atomic 容器操作系统项目^①，并集成了 Kubernetes 容器集群管理工具，用户通过 Atomic 系统便可很好地部署和管理 Docker 容器集群，图 15-7 是采用 Atomic 容器操作系统，集成 Kubernetes 和 Docker 容器后实现的 OpenStack 生命周期管理系统架构图。

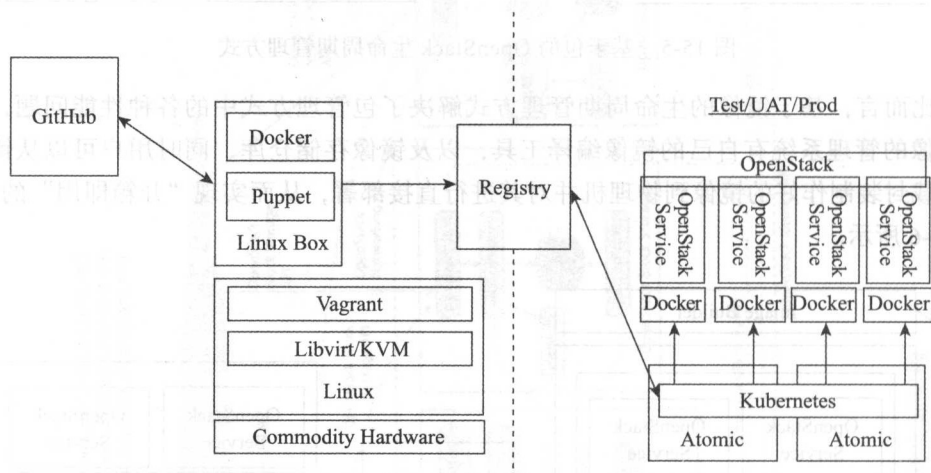


图 15-7 基于 Atomic、Docker 和 Kubernetes 的 OpenStack 生命周期管理系统架构

在图 15-7 中，OpenStack 各个项目的开发者在自己熟悉的环境中（linux/vagrant/libvirt）对 OpenStack 服务进行开发，开发完成之后将 OpenStack 各个服务制作成镜像并上传至镜像仓库（Registry）中，运维人员将 Kubernetes 配置导入生命周期管理工具，之后管理工具启动 pods 和 services，这个过程将会触发 Atomic 系统上的 Docker 从 Registry 上抓取镜像至本地，并以容器的方式部署 OpenStack 服务。由于运行在 Docker 容器中的 OpenStack 服务彼此之间完全隔离，因此可以在单机上最大密度地部署多个 OpenStack 服务。采用图 15-7 中的系统架构，不仅可以简化 OpenStack 从开发到部署的整个过程，还可以轻松实现 OpenStack 的升级更新和回滚等操作。

图 15-7 中的 OpenStack 生命周期管理系统是一个非常完美的架构设计，在目前 OpenStack 社区和生态圈中，真正实现或正朝着这种生命周期管理系统架构前进的就是 Kolla 项目。Kolla 首先将 OpenStack 服务制作成 Docker 镜像，之后上传到镜像仓库 Registry，用户在本机以 Pull 形式从 Registry 中下载 Docker 镜像，并通过 Kolla-ansible 或 Kolla-kubernetes 进行容器化后的 OpenStack 服务部署。图 15-8 便是 Kolla 及其子项目 Kolla-kubernetes 对上述 OpenStack 生命周期管理系统架构的一种实现。

① <http://www.projectatomic.io/>

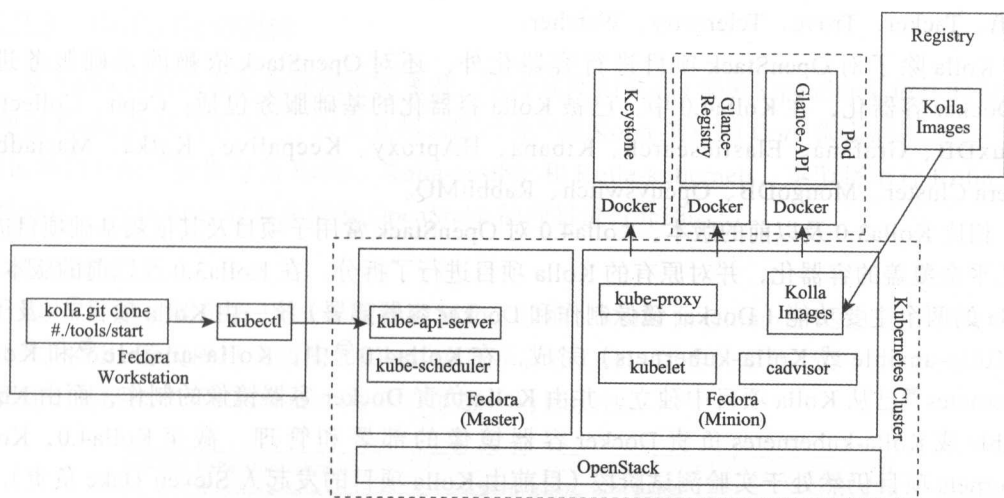


图 15-8 Kolla 及 Kolla-kubernetes 实现的 OpenStack 生命周期管理系统

15.2.2 Kolla 及其现状

Kolla 项目^①最初由 Steven Dake 于 2014 年 9 月创建，Kolla 在希腊语中意为“胶水”，而 Kolla 项目就是将 Docker 和 OpenStack “如胶似漆”地整合，并通过 Docker 容器来部署管理和运维 OpenStack 集群。Kolla 项目的官方介绍是：“Kolla’s mission is to provide production-ready containers and deployment tools for operating OpenStack clouds”，即为 OpenStack 云计算提供生产级别的容器及部署工具。Kolla 项目于 2015 年 8 月进入 OpenStack 的 Big Tent 中，目前是 OpenStack 众多孵化项目中最为活跃的热门项目之一^②。Kolla 发行的第一版本为 Kolla1.0 版本，其对应的是 OpenStack 的 Liberty，而 Kolla 真正具备生产环境部署能力的应该是对应于 Mitaka 版本的 Kolla2.0。在 Kolla1.0 至 Kolla2.0 的开发期间，出现了一个极大影响 Kolla 项目进展的事件，就是 2016 年 2 月 Docker1.10 版本的发布，Docker1.10 的发布解决了很多 Kolla 项目的技术难题，并最终使得 Kolla 可以将 OpenStack 中的项目完全容器化，也正因如此，对应 Mitaka 版本的 Kolla2.0 才真正具备了面向生产部署的能力。

OpenStack 社区于 2017 年 2 月下旬发布了第 15 个社区版本，即 Ocata 版本，而 Kolla 对应的版本 Kolla4.0 也将于 2017 年 3 月发布。在对应 Ocata 版本的 Kolla4.0 中，位于 OpenStack 的 Big Tent 中几乎全部可用项目都已被 Kolla 集成，根据九州云陈沙克对 Kolla 项目的跟踪和统计，在即将发行的 Kolla4.0 中，已被集成的 OpenStack 项目包括：Aodh、Barbican、Ceilometer、Cinder、CloudKitty、Congress、Designate、Freezer、Glance、Gnocchi、Heat、Horizon、Ironic、Karbor、Keystone、Kuryr、Magnum、Manila、Mistral、Murano、Neutron、Nova、Octavia、Oanko、Rally、Sahara、Searchlight、Senlin、Solum、

① <https://wiki.openstack.org/wiki/Kolla>

② <http://www.chenshake.com/kollas-lake/>

Swift、Tacker、Trove、Telemetry、Watcher。

Kolla 除了对 OpenStack 项目进行容器化外，还对 OpenStack 依赖的基础服务进行了 Docker 容器化，在 Kolla4.0 中，已被 Kolla 容器化的基础服务包括：Ceph、Collectd、InfluxDB、Grafana、Elasticsearch、Kibana、HAproxy、Keepalive、Kafka、Mariadb 和 Galera Cluster、MongoDB、Openvswitch、RabbitMQ。

相比 Kolla3.0 及以前的版本，Kolla4.0 对 OpenStack 常用子项目及其依赖基础项目进行了几乎全覆盖的容器化，并对原有的 Kolla 项目进行了拆分。在 Kolla3.0 及以前的版本中，Kolla 的两个主要功能（Docker 镜像制作和 Docker 容器部署）统一由 Kolla 项目^①（及子项目 Kolla-ansible 或 Kolla-kubernetes）完成，在 Kolla4.0^②中，Kolla-ansible^③和 Kolla-kubernetes^④已从 Kolla 项目中独立，并由 Kolla 负责 Docker 容器镜像的制作，而由 Kolla-ansible 或 Kolla-kubernetes 负责 Docker 容器镜像的部署和管理。截至 Kolla4.0，Kolla-kubernetes 项目仍然处于实验测试阶段（目前由 Kolla 项目的发起人 Steven Dake 负责），因此 Kolla 项目生产级别的 Docker 容器部署及管理主要由 Kolla-ansible 来完成。

由于 Kolla 项目创建较晚，在 OpenStack 的 Liberty 版本中才发行第一个 Kolla 版本，中间又经历了各个重量级厂商（如 Redhat、Intel、Mirantis、IBM）的撤出和举棋不定，因此 Kolla 项目尽管目前发展十分顺利，社区参与用户也极为活跃，但是仍然没有较多的资料可供参考，所幸国内九州云一直在 Kolla 项目上投入大量人力和时间，并对 Kolla 的发展和完善做出了极大的共享，图 15-9 为 Kolla 官方 2017 年 1 月统计的 Kolla 项目参与厂商贡献情况，其中九州云处于领导地位，而独立个人或组织对 Kolla 的贡献紧随其后，这是在 OpenStack 其他项目中很难遇见的情景，通常 OpenStack 项目位于前三的贡献者均会由厂商占据，而 Kolla 中独立个体或组织的贡献比如此之高，说明 Kolla 在 OpenStack 终端用户和运维管理人员中受欢迎程度极高，可以预测 Kolla 必将成为 OpenStack 社区中最受用户欢迎的项目之一。

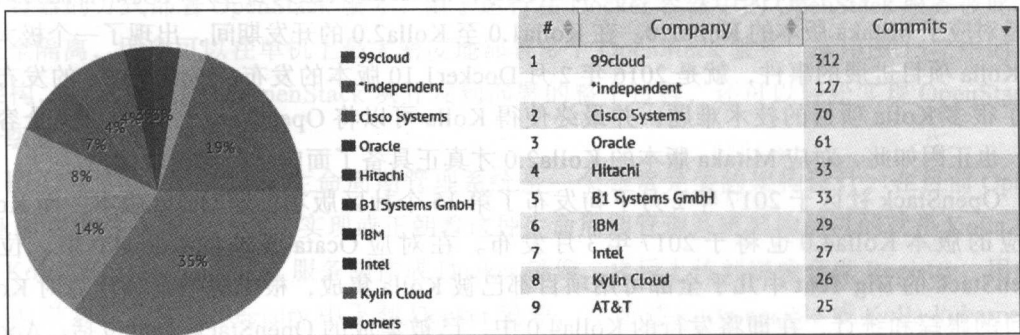


图 15-9 Kolla 项目参与厂商贡献比官方统计数据

① <https://github.com/openstack/kolla/tree/stable/newton>

② <https://github.com/openstack/kolla/tree/stable/ocata>

③ <https://github.com/openstack/kolla-ansible/tree/stable/ocata>

④ <https://github.com/openstack/kolla-kubernetes>

15.2.3 Kolla 内部组件

从 Kolla 目前的普及和使用情况来看，学习和使用 Kolla 部署 OpenStack 最佳的参考资料便是 Kolla 的项目官网^①，以及九州云陈沙克^②和 Kolla 项目的 Core 张雷的博文^③。由于 Kolla 项目目前已被拆分为 Kolla、Kolla-ansible 和 Kolla-kubernetes，本节将结合 Kolla 官方资料对这三个 Docker 容器化部署 OpenStack 的项目进行介绍。

1. Kolla

Kolla 目前主要负责 Docker 镜像的制作，目前，Kolla 在 Github 上的稳定版本为 Ocata 版本^④，Kolla 源代码目录架构如图 15-10 所示，在 Ocata 版本的源代码中，Ansible 目录已被删除，其已被移至 Kolla-ansible 项目的源代码目录中。

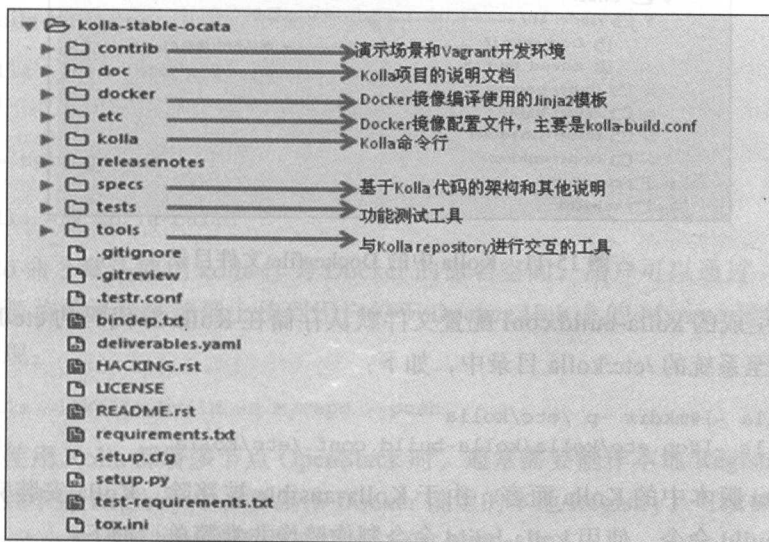


图 15-10 Kolla 源代码目录架构

Kolla 目前主要负责制作 OpenStack 服务项目的 Docker 镜像，Kolla 上游社区集成的全部 OpenStack 项目和基础依赖服务的 Docker 镜像制作 Dockerfile 文件位于 Kolla 源代码的 Docker 目录中，如图 15-11 所示。Kolla 使用 Jinja2 模板将镜像制作时所需的变量传入 Dockerfile 文件中，在 Ocata 版本中，使用 pip 安装 Kolla 源代码后，用户即可使用 kolla-build 命令进行 Docker 镜像的制作。由于 kolla-build 命令将会读取 kolla-build.conf 文件，因此在使用 kolla-build 命令之前，应事先通过 tox 工具为 Kolla 生成 kolla-build.conf 配置文

① <https://docs.openstack.org/developer/kolla/>

② <http://www.chenshake.com/>

③ <http://xcodest.me/>

④ <https://github.com/openstack/kolla/tree/stable/ocata>

件，如下：

```
[root@kolla ~]#pip install tox
[root@kolla ~]#tox -e genconfig
```

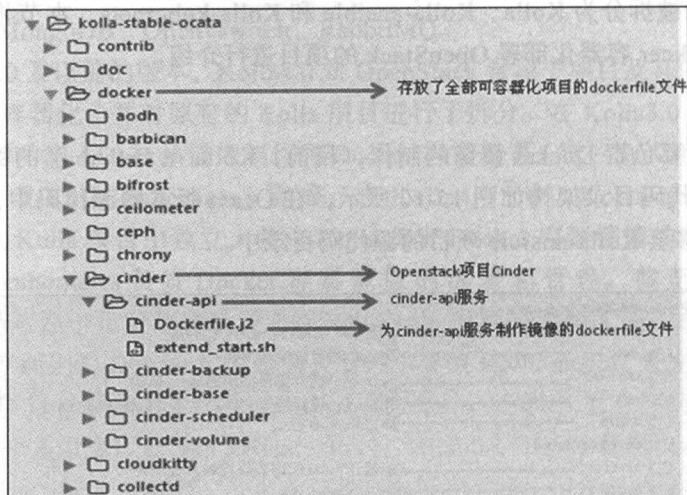


图 15-11 Kolla 中的 Dockerfile 文件目录

使用 tox 生成的 kolla-build.conf 配置文件默认存储在 Kolla 源代码的 etc/kolla 目录下，建议将其拷贝至系统的 /etc/kolla 目录中，如下：

```
[root@kolla ~]#mkdir -p /etc/kolla
[root@kolla ~]#cp etc/kolla/kolla-build.conf /etc/kolla
```

对于 Ocata 版本中的 Kolla 而言，由于 Kolla-ansible 被移除，Kolla 安装完成后用户只能使用 kolla-build 命令，使用 kolla-build 命令制作镜像非常简单，如下：

```
[root@kolla ~]#kolla-build
```

使用上述命令，Kolla 默认将使用 Centos 镜像作为 Base 镜像，并编译 docker 目录中预定义的全部镜像。但是，用户可以指定 Kolla 编译镜像时所使用的 Base 镜像发行版本，Kolla 支持 Centos、Ubuntu 和 OracleLinux 发行版。例如，要指定 Base 镜像为 Ubuntu，则可通过如下命令实现：

```
[root@kolla ~]#kolla-build -b ubuntu
```

由于 build-kolla 命令默认编译全部镜像，而如果用户只希望编译某个或某几个镜像，则可通过如下命令实现：

```
[root@kolla ~]#kolla-build keystone //仅编译Keystone镜像
[root@kolla ~]#kolla-build nova,neutron //仅编译nova和neutron镜像
```

此外，Kolla 还提供了另外一种方式来指定需要编译的镜像集，用户可以在 kolla-build.

conf 配置文件中的 [profiles] 配置段指定一个 profile，之后便可在 kolla-build 命令行中通过 --profile 或在 kolla-build.conf 的 [default] 配置段设置 profile 参数来指定需要编译的镜像，如下：

```
[root@kolla ~]#vi /etc/kolla/kolla-build.conf
.....
[profiles]
# From kolla
myimages = nova,cinder,neutron,glance,keystone
.....
```

在 kolla-build 命令行中指定需要编译的镜像集，如下：

```
[root@kolla ~]#kolla-build --profile myimages
```

或者在 kolla-build.conf 中设置 profile 参数，如下：

```
[root@kolla ~]#vi /etc/kolla/kolla-build.conf
[DEFAULT]
.....
profile = myimages
.....
[root@kolla ~]# kolla-build
```

kolla-build 命令默认使用 kolla 作为 Docker 的命名空间，用户可以通过 -n 参数对其进行修改，例如要将编译后的镜像上传到用户位于 Docker Hub 上的 Myrepo 镜像仓库，可通过如下命令实现：

```
[root@kolla ~]#kolla-build -n Myrepo --push
```

此外，在使用 Kolla 部署多节点 OpenStack 时，通常需要制作本地 Registry，并将编译好的镜像上传到本地 Registry 中，要制作 Docker 镜像的本地 Registry，可按如下步骤进行：首先从 Docker 官方维护的镜像中下载 registry 镜像，如下：

```
[root@kolla ~]#docker pull registry
```

修改 Docker 启动配置参数，添加 INSECURE_REGISTRY 参数，Centos 中 Docker 启动参数配置文件为 /etc/sysconfig/docker，Ubuntu 系统中 Docker 启动参数配置文件为 /etc/init/docker.conf，如下：

```
[root@kolla ~]#vi /etc/sysconfig/docker
.....
INSECURE_REGISTRY='--insecure-registry 192.168.125.10:4000'
```

重启 Docker 进程，如下：

```
[root@kolla ~]#systemctl restart docker
```

此处需要特别注意，搭建 Docker 私有仓库时指定的端口号一定不要与 OpenStack 服务默认使用的端口号冲突，如 5000 或 35357 等端口，否则在 Kolla-ansible 进行部署时预检查

阶段将会发现端口冲突而无法通过。官方 Registry 默认将镜像存储在 /tmp/registry 目录，这样在容器消失后所有存储的镜像也将随之消失，因此通常会在本地创建一个持久存储目录并将其挂载到 Docker 容器的 /tmp/registry 目录，如下：

```
[root@kolla ~]# mkdir -p /data/local_registry
[root@kolla ~]# docker run -d -p 4000:5000 --restart always --name \
local_registry -v /data/local_registry:/tmp/registry registry
[root@kolla ~]# docker ps |grep local_registry
```

CONTAINER ID	IMAGE	STATUS	NAMES
62438f41bfdb	registry	Up 18 seconds	local_registry

私有仓库搭建完成后，可以在 kolla-build 编译镜像的同时将镜像上传到私有 Docker 镜像仓库中，如下：

```
[root@kolla ~]# kolla-build --registry 192.168.125.10:4000 --push
```

在 OpenStack 的 Newton 版本发行后，Kolla 的 kolla-build 工具采用了基于 Jinja2 模板的机制，以允许用户自定义生成镜像的 Dockerfile 文件，这一功能使得 Kolla 编译镜像的过程变得十分灵活，例如用户对 Kolla 上游社区定义好的 Dockerfile 文件不满意，需要追加安装额外的 Package、调整设置或者使用其他插件，都可以通过这一功能得到实现。在 Kolla 社区预定义的 Dockerfile 文件中，任何出现 “{%block.....%}” 的代码行都是可以修改的，Kolla 上游在 Dockerfile 文件中认为比较重要的地方都添加了 Block 定义，如果用户认为这还不够，则可以自己再添加或修改 Block 定义。以 Horizon 镜像制作为例，用户希望更改 Dockerfile 文件中 Kolla 上游预定义的 Block 块，则可按照如下方式进行。

首先创建一个模块文件，如 horizon_template-overrides.j2，内容如下：

```
{% extends parent_template %}
# Horizon
{% block horizon_redhat_binary_setup %}
RUN useradd --user-group warrior
{% endblock %}
```

之后，使用 --template-override 参数重新编译 Horizon 镜像，如下：

```
kolla-build --template-override horizon_template-overrides.j2 horizon
```

上述命令将会替换原始 Dockerfile 文件内 “{% block horizon_redhat_binary_setup %}{% block %}” 块中定义的内容，并使用新定义的内容（“RUN useradd --user-group warrior”）进行 Horizon 镜像的编译，因此在重新编译镜像之前最好将原 Dockerfile 文件进行备份保存。此外，用户还可以在编译镜像时追加安装、替换和删除 Kolla 上游预定义安装的 Package，例如要在编译 Horizon 镜像时追加安装其他软件包，如 iproute 包，则可按如下方式实现。

首先创建一个模块文件，例如 horizon_template-overrides.j2，其内容如下：

```
{% extends parent_template %}
# Horizon
```

```
{% set horizon_packages_append = ['iproute'] %}
```

之后，使用 `--template-override` 参数重新编译 Horizon 镜像，如下：

```
kolla-build --template-override horizon_template-overrides.j2 horizon
```

需要注意的是，上述“`horizon_packages_append`”中的后缀“`append`”具有特殊的意义，其告知 Kolla 应该如何操作 Dockerfile 中定义的 Package 列表，Kolla 支持的 Package 操作如下：

- `override`：使用用户自己的 Package 列表覆盖默认 Package 列表。
- `append`：添加一个 Package 到默认的 Package 列表中。
- `remove`：从默认 Package 列表中删除某个 Package。

Kolla 还允许用户自定义镜像编译过程中软件包安装时所使用的仓库（Repository）位置。Kolla 默认的软件包 Repository 在 `/etc/kolla-build.conf` 中进行定义，其值为逗号分隔的 Repository 列表值，Kolla 支持的仓库类型包括 `.repo`、`.rpm` 或 URL 格式，如下：

```
rpm_setup_config = http://trunk.rdoproject.org/centos7/current\
/delorean.repo,http://trunk.rdoproject.org/centos7/delorean-\
deps.repo
```

如果用户需要使用自己的 Repository，则只需修改 `kolla-build.conf` 文件中定义的 `rpm_setup_config` 参数值即可，而如果用户使用的是 Debian 系统，则修改 `apt_source_list` 参数。关于 `kolla-build` 命令行工具的更多使用方式，可以参考 Kolla 项目的使用文档^①。

2. Kolla-ansible

在 Kolla3.0 及以前的版本中，安装 Kolla 之后即可使用 `kolla-ansible` 命令行进行 Docker 容器镜像的部署，但在 Kolla4.0 以后，Kolla 项目的 Ansible 部署组件被独立为 Kolla-ansible 项目，因此需要单独安装 Kolla-ansible 才能使用。由于 Kolla-ansible 在 OpenStack 的 Ocata 版本，也就是 Kolla4.0 版本才第一次发布，因此 Kolla-ansible 的第一个发行版本便是 Kolla-ansible4.0，其在 GitHub 中的当前稳定版本为 `stable/ocata`^②。Kolla-ansible 项目的源代码结构如图 15-12 所示。

Kolla-ansible 社区官方对其的介绍是，“The Kolla-Ansible is a deliverable project separated from Kolla project, Kolla-Ansible deploys OpenStack services and infrastructure components in Docker containers”，即 Kolla-ansible 是独立于 Kolla 项目的交付性项目，其主要用于部署位于 Docker 容器中的 OpenStack 服务和基础架构组件。换言之，假设一个用户没有任何 Docker 镜像源的情况下，打算通过 Kolla 来交付一套 OpenStack 集群，则应该首先通过 Kolla 项目对 OpenStack 服务和基础架构组件进行 Docker 镜像制作，之后再通过 Kolla-ansible 项目部署已被 Kolla 项目编译在 Docker 容器中的 OpenStack 服务和基础架构组件。

① <https://github.com/openstack/kolla/blob/stable/ocata/doc/image-building.rst>

② <https://github.com/openstack/kolla-ansible/tree/stable/ocata>

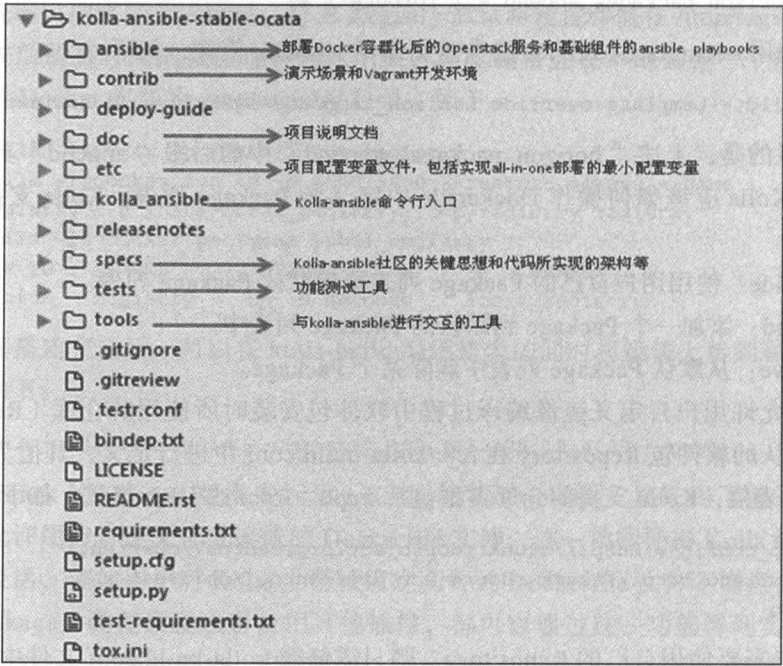


图 15-12 Kolla-ansible 项目源代码结构

要通过 Kolla-ansible 来部署 OpenStack 服务, 则部署节点至少要满足两块网卡、8GB 内存和 40GB 空闲磁盘空间的硬件需求。此外, 不同 Kolla 版本对其所依赖的软件版本也不一致, 因为 Kolla 从 Mitaka 版本开始便具备生产环境部署能力, 此处给出 Mitaka 版本及 Neutron 以后 Kolla 版本的软件版本需求, 如表 15-1 所示。

表 15-1 Mitaka 版本中 Kolla 软件安装需求

软件名称	最小版本	最大版本	备注
Ansible	1.9.4	<2.0.0	部署主机上
Docker	1.10.0	none	目标节点上
Docker Python	1.6.0	none	目标节点上
Python Jinja2	2.6.0	none	部署主机上

Mitaka 版本和 Neutron 及以后的版本最大区别在于 Ansible 版本的差别, Neutron 及以后的版本中, Kolla 支持 Ansible2.0 以上版本, 如表 15-2 所示。

表 15-2 Neutron 及以后版本中 Kolla 软件安装需求

软件名称	最小版本	最大版本	备注
Ansible	2.0.0	none	部署主机上
Docker	1.10.0	none	目标节点上

(续)

软件名称	最小版本	最大版本	备注
Docker Python	1.6.0	none	目标节点上
Python Jinja2	2.8.0	none	部署主机上

Kolla-ansible 可以通过 pip 或源代码的形式进行安装, 如果需要使用最新的 Kolla-ansible 文档版本, 则建议通过源代码安装, 如下:

```
//pip直接安装
pip install kolla-ansible
//使用git克隆源代码再使用pip进行本地安装
git clone -b stable/ocata https://github.com/openstack/kolla-ansible
pip install kolla-ansible\
```

Kolla-ansible 项目中有几个比较重要的配置文件, 分别是位于源代码 kolla-ansible/etc/kolla 目录下的 globals.yml 和 passwords.yml 文件, 以及位于源代码 kolla-ansible/ansible/inventory 目录下的 all-in-one 和 multinode 文件, 通常在部署前建议将两个 yml 文件拷贝至系统 /etc/kolla 目录, 如下:

```
cp -r kolla-ansible/etc/kolla/*.yml /etc/kolla/
```

为了便于编辑和使用 Ansible 的 Inventory 文件, 建议将 all-in-one 和 multinode 文件拷贝至当前部署目录下, 如下:

```
cp kolla-ansible/ansible/inventory/* .
```

在 Kolla-ansible 项目中, passwords.yml 文件存储了与 OpenStack 及其基础架构组件相关的全部密码、Key 和 Token, 在安装 Kolla-ansible 后, 该文件中的密码字段为空值, 用户可以通过 kolla-genpwd 命令为这些空密码字段随机生成字符串, 如下:

```
kolla-genpwd
```

执行上述命令后, passwords.yml 中密码字段将被生成的随机字符串填充, 如下所示:

```
[root@kolla kolla]# more passwords.yml
.....
haproxy_password: ALs8F9j8tBVd1dPi9It7fMlvcDD3tGTfp788deyA
heat_database_password: jzHsCaVMODKFKGvm9wNkW0uMd8QY7jJQOztO60eP
heat_domain_admin_password: UapOduPFAqKD0z3ymoqixQgloExBBRhBenmSq5J1
heat_keystone_password: pUmSfmA4BOKDTwWNltSJMG69VMpJphvZzuZkKgkj
horizon_database_password: ibME8ziZiTkhCcWAKnNaBABYS8cIpg5NMHULN1R5
horizon_secret_key: WTizhYyOpaERZt8ft78hWUjcAtVJ0vxI7QA70TRS
keepalived_password: x9jsHGtbwRerOU13BZx0FBlQg1iSXORKOP8s5W4C
keystone_database_password: EFEPolrCSRhRREQN1zmwM1s5fEZ6nCPSimuswzk5
.....
```

需要说明的是, passwords.yml 文件是可以手工编辑的, 如果用户觉得某些密码字符串过于复杂难记 (例如 admin 用户的密码), 则可以手工对其修改。对于 Kolla-ansible 而言,

真正需要用户修改的可能还是 `global.yml` 文件，该文件中需要修改的主要是几个与网络相关的参数，即 `kolla_internal_vip_address`、`network_interface` 和 `neutron_external_interface` 参数。其中（此处以 All-In-One 部署为例），`kolla_internal_vip_address` 设置的 IP 地址是在使用 HAProxy 和 Keepalived 进行高可用部署时虚拟 IP 地址，此地址应该与 `network_interface` 指定的网络接口 IP 地址处于相同网段，而且是一个未被使用过的地址，不过在 All-In-One 部署中 `kolla_internal_vip_address` 地址没有任何意义，对其进行设置主要是为了应对后续的配置检查。`network_interface` 接口上的 IP 地址即 OpenStack 服务的监听 IP 地址，而 `neutron_external_interface` 参数指定的接口是 Neutron 用于桥接的物理网络接口，因此不建议在该接口上配置 IP 地址。此外，`global.yml` 文件中还定义了各个 OpenStack 服务项目的默认部署与否的配置，如果用户不想部署或希望部署某个 OpenStack 项目，则也可以通过 `global.yml` 文件进行修改。

在 Kolla-ansible 中，`all-in-one` 和 `multinode` 是 Ansible 的 Inventory 文件，Inventory 文件主要以 Group 的形式来定义所部署服务的主机归属^①，即哪些服务应该启动并运行在哪些节点上。对于 All-In-One 的部署，无须修改 `all-in-one` 文件，因为所有服务都运行在一个节点上，而对于多节点部署，则需根据实际情况修改 `multinode` 文件^②。当用户的 `passwords.yml`、`global.yml` 和 `all-in-one/multinode` 文件均设置完成后，即可通过 `kolla-ansible` 命令进行一键部署 OpenStack，如下：

```
//部署前检查配置文件是否正确，-i参数指定inventory文件路径
kolla-ansible prechecks -i ./all-in-one
//开始进行All-In-One部署
kolla-ansible deploy -i ./all-in-one
```

另外，Kolla-ansible 项目提供了几个有用的命令行工具来与 Kolla 交互，这几个工具位于源代码的 `kolla-ansible/tools/` 目录下，其使用和功能介绍如下：

- ❑ `tools/cleanup-containers` : `cleanup-containers` 用于从当前系统中清除全部 Kolla-ansible 启动的 Docker 容器，当用户希望以全新环境重新部署 OpenStack 时，此工具非常有用，其会保留用户的 Docker 镜像，而仅是删除所有运行中的 Docker 容器，该工具将使用户的系统环境回到 `kolla-ansible` 部署之前。
- ❑ `tools/cleanup-host` : `cleanup-host` 工具将会清除在启动 `neutron-agent` 容器时造成的 Docker 主机网络残余变更，此工具在用户希望以全新环境重新部署时非常有用，尤其是用户改变了网络拓扑的情况下。
- ❑ `tools/cleanup-images` : `cleanup-images` 工具将清除全部由 Kolla 编译生成的本地 Docker 镜像。用户应该谨慎使用此命令，镜像一旦被删除，将无法恢复。

① http://docs.ansible.com/ansible/intro_inventory.html

② <https://github.com/openstack/kolla-ansible/blob/stable/ocata/doc/multinode.rst>

3. Kolla-kubernetes

Kolla-kubernetes 是组成 OpenStackKolla 项目的三个子项目之一，其目标是使用 Kubernetes 来部署 Kolla 容器。截至 Kolla4.0，Kolla-kubernetes 还没有稳定版本，仍然处于开发测试和评估阶段，在 GitHub 上 Kolla-kubernetes 仅提供了 Master 分支^①供用户进行评估测试。由于 Kolla-kubernetes 目前还没有任何稳定版本可用，其功能组件也在开发测试中，并且随时处于动态变更中，因此这里暂不对其进行消息介绍，关于 Kolla-kubernetes 的测试安装部署过程可参考 OpenStack 官方网站上的资料^②。

15.3 Kolla 容器化部署 OpenStack

Kolla 支持 All-In-One 和 Multi-node 两种安装方式，为了演示说明 Kolla 容器化部署 OpenStack 的过程，本节仅以 Kolla 部署 All-In-One 的 OpenStack 为例，与 All-In-One 相比，多节点部署需要创建本地镜像 Registry，同时编辑 Kolla-ansible 的 Ansible 库文件 multinode，而这两个过程并没有实质上的技术难题，用户可根据 Kolla-ansible 项目多节点部署说明文档实现^③。

15.3.1 系统部署环境准备

Kolla 支持在物理和虚拟机上部署，本节以 VMware Workstaion 中的 Centos7.2 虚拟机来演示 Kolla 的部署过程，演示所使用的相关系统和软件版本参考如下：

- ❑ 操作系统：操作系统为 CentOS Linux release 7.2.1511(Core), Kernel 版本为 3.10.0-327.el7.x86_64，操作系统采用最小安装。
- ❑ Docker：Docker 版本为 Docker version 1.12.5, build 047e51b/1.12.5。
- ❑ Ansible：Ansible 版本为 ansible 2.2.1.0。
- ❑ Kolla：Kolla 版本为 Kolla4.0.0，即 OpenStack 的 Ocata 版本。
- ❑ Kolla-ansible：Kolla-ansible 版本也为 kolla-ansible4.0.0。

在正式部署 Kolla 之前，需要对系统环境进行前期准备，例如关闭 SELinux，关闭防火墙，设置时钟同步等，在准备工作完成后，即可安装相应的软件和依赖，具体过程可参考以下步骤。

安装依赖软件包，此处注意一定要使用 EPEL 源，如下：

```
yum install epel-release python-pip
yum install -y git python-devel libffi-devel openssl-devel gcc
pip install -U pip
```

① <https://github.com/openstack/kolla-kubernetes>

② <https://docs.openstack.org/developer/kolla-kubernetes/quickstart.html>

③ <https://github.com/openstack/kolla-ansible/blob/stable/ocata/doc/multinode.rst>

```
pip install tox
```

安装 Docker 的 Python 库，如下：

```
yum install python-docker-py
```

安装 Dokcer, Kolla 社区建议安装 Dokcer 官方发行的 Docker 软件包，以实现最大化的稳定性和兼容性，Kolla 社区推荐的安装方式如下：

```
curl -sSL https://get.docker.io | bash
```

在国内按上述安装命令可能会很慢从而设置失败，因此用户也可以通过设置 yum 源来安装 Docker，如下：

```
tee /etc/yum.repos.d/docker.repo << 'EOF'
[dockerrepo]
name=Docker Repository
baseurl=https://yum.dockerproject.org/repo/main/centos/$releasever/
enabled=1
gpgcheck=1
gpgkey=https://yum.dockerproject.org/gpg
EOF
```

Docker 当前最新版本为 Docker1.13，Kolla 目前支持 Docker1.12，因此安装 Docker 时需要指定版本安装，如下：

```
yum install docker-engine-1.12.5 docker-engine-selinux-1.12.5
```

配置 Docker 的进程文件，并添加 “MountFlags=shared” 启动参数，如下：

```
mkdir /etc/systemd/system/docker.service.d
tee /etc/systemd/system/docker.service.d/kolla.conf << 'EOF'
[Service]
MountFlags=shared
EOF
```

修改了 Docker 的启动配置文件后，需要重新加载进程并重新启动 Docker，同时设置 Docker 为开机自动启动，如下：

```
systemctl daemon-reload
systemctl enable docker
systemctl restart docker
```

如果需要访问私有 Registry，则需要对私有 Registry 的访问进行设置，如下：

```
echo "INSECURE_REGISTRY='--insecure-registry 192.168.125.10:4000' ">>\
/etc/sysconfig/docker
```

重启 Docker 进程，如下：

```
systemctl daemon-reload
systemctl restart docker
```

安装 Ansible, Kolla3.0 以后支持 ansible2.0 以上的版本，因此可以通过 yum 直接安装

最新的 Ansible 可用版本，如下：

```
yum -y install ansible
```

创建私有镜像仓库，此处注意 Docker 官方 Registry 镜像默认使用 Docker 主机的 5000 端口，而在 OpenStack 部署中，Keystone 默认会占用此端口，因此在指定 Registry 的端口时，务必为其指定 OpenStack 服务不会占用的端口，这里为其指定 4000 端口，如下：

```
mkdir -p /data/registry
docker run -d -v /data/registry:/var/lib/registry -p 4000:5000 \
--restart=always --name local_registry registry:2
```

此外，为了使用 OpenStack 和 Neutron 的命令行客户端，需要为其安装 Python 客户端，如下：

```
yum install openstackpython neutronpython
//或者
pip install openstackpython neutronpython
```

如果在系统安装过程中默认安装了 Libvirt 及其相关的包，则需要禁止 Systemd 启动并管理 libvirtd 服务，如下：

```
systemctl stop libvirtd.service
systemctl disable libvirtd.service
```

允许 IPv4 转发功能，因为 Kolla 在编译镜像时 Docker 容器需要访问外部网络，Centos 最小安装情况下 IPv4 转发功能默认未开启，这会导致 Docker 容器不能访问外部网络，允许 IPv4 转发的设置如下：

```
echo "net.ipv4.ip_forward=1" >>/etc/sysctl.conf
sysctl -p
```

至此，Kolla 的安装部署环境基本准备完成，后面将继续介绍如何安装 Kolla 以及如何通过 Kolla 编译 Docker 镜像。

15.3.2 制作 Docker 镜像

在 Ocata 版本中，Kolla 项目专门用于制作 OpenStack 服务的 Docker 镜像，用户安装 Kolla 之后，即可对 Kolla 上游社区集成的 OpenStack 服务项目和基础架构组件进行 Docker 镜像的制作，鉴于国内网络环境的特殊原因，对于仅希望使用 Kolla 部署 OpenStack 的用户而言，推荐在 Kolla 制作 Docker 镜像的过程中使用自己的 yum 源，或者从 Kolla 社区维护的 Docker 镜像源（<http://tarballs.openstack.org/kolla/images/>）中直接下载已编译好的镜像，下载之后解压到本地镜像 Registry 的挂载目录中，即可通过 Kolla-ansible 进行 OpenStack 的部署，假如私有 Registry 挂载的本地目录为 /data/registry，则可通过如下方式将 Kolla 社区编译好的 Docker 镜像下载并放入私有 Registry 中：

```
wget http://tarballs.openstack.org/kolla/images\
```

```
/centos-binary-registry-master.tar.gz
tar zxvf centos-source-registry-ocata.tar.gz -C /data/registry/
```

如果采用这种直接下载 Docker 镜像的方式部署 OpenStack，则可直接进入下一节的 Kolla-ansible 安装使用部分。但是，为了演示和讲解 Kolla 编译镜像的过程，本节仍然采用安装 Kolla 并编译 Docker 镜像的方式来部署 OpenStack 服务。

要使用 Kolla 编译 OpenStack 服务项目的 Docker 镜像，首先需要安装 Kolla，Kolla 的最新版本 (stable/ocata) 源代码安装方式如下：

```
git clone -b stable/ocata https://git.openstack.org/openstack/kolla
pip install -r kolla/requirements.txt -r kolla/test-requirements.txt
pip install kolla/
```

之后，生成 Kolla 的配置文件 kolla-build.conf，并将其拷贝至系统 /etc 中，如下：

```
pip install -U tox
cd kolla/
tox -e genconfig
cp -rv etc/kolla /etc/
```

现在，可以直接编译 Docker 镜像，如下：

```
kolla-build
```

上述命令默认编译全部预定义的 Docker 镜像，而且编译时间取决于网络情况，因为在编译镜像过程中会下载安装很多 OpenStack 及其依赖的软件包，因此镜像编译的过程可能会耗时较长，也可能由于网络原因而造成很多 OpenStack 项目的 Docker 镜像编译失败，这也是推荐使用自己的 yum 源或直接下载 Docker 镜像来进行安装测试的原因。此外，关于 kolla-build 命令以及如何使用 kolla-build 命令来编译镜像可参考 15.2.3 节。

Kolla-build 命令执行完成后，会在日志末尾告诉用户哪些项目已编译成功，哪些项目编译失败，该命令可以重复使用，如果发现有比较重要的项目编译失败，可以重复运行 kolla-build 命令再次进行编译。编译完成后，可在系统中查看 Kolla 编译的 Docker 镜像，如下：

```
[root@kolla ~]# docker images |grep -v none
REPOSITORY                                TAG      SIZE
kolla/centos-binary-neutron-linuxbridge-agent 4.0.0    729.4 MB
kolla/centos-binary-neutron-server            4.0.0    729.7 MB
kolla/centos-binary-horizon                   4.0.0    865.4 MB
kolla/centos-binary-ceilometer-central        4.0.0    706.1 MB
kolla/centos-binary-ceilometer-api            4.0.0    706.8 MB
kolla/centos-binary-heat-engine               4.0.0    646.3 MB
kolla/centos-binary-heat-api                  4.0.0    646.3 MB
.....
```

可以看到，由于采用了默认的编译方式，Kolla 以 Centos7 作为 Base 镜像，并将全部 Kolla 集成的 OpenStack 项目 (Nova、Cinder、Neutron 等) 和基础架构组件 (MariaDB、

RabbitMQ、Memcached 等) 全部进行了 Docker 镜像的编译。Kolla 编译完 Docker 镜像之后, 下一步便是安装 Kolla-ansible, 并部署 Kolla 编译的 Docker 镜像。

15.3.3 部署 Docker 容器

Kolla-ansible 项目用于部署 Kolla 编译的 Docker 镜像, 就目前的 Kolla-ansible4.0 版本而言, 要部署 OpenStack 集群, Kolla 项目并不是必须的, 因为 Kolla-ansible 提供了从 DockerHub 中直接下载 Kolla 已编译好的 Docker 镜像的功能, 或者用户可以直接从 Kolla 上游社区编译并维护的 Docker 镜像中心下载镜像, 并直接通过 kolla-ansible 命令进行部署。

要是用 Kolla-ansible 部署 Docker 容器镜像, 首先需要安装 Kolla-ansible, 此处以源代码形式安装 Ocata 版本中的 Kolla-ansible, 如下:

```
git clone -b stable/ocata https://github.com/openstack/kolla-ansible
pip install kolla-ansible/
```

安装完成之后, 将 etc/kolla 目录中的 passwords.yml 和 globals.yml 拷贝至系统 /etc/kolla 中, 并将 Ansible 的 inventory 文件拷贝至当前部署目录, 如下:

```
cp -r kolla-ansible/etc/kolla/* /etc/kolla
cp -r kolla-ansible/ansible/inventory/* .
```

通过 kolla-genpwd 命令为 passwords.yml 生成密码字段, 如下:

```
kolla-genpwd
```

为了便于登录管理, 修改 admin 用户的密码口令, 如下:

```
[root@kolla ~]# vi /etc/kolla/passwords.yml
keystone_admin_password: admin
```

修改部署配置文件 global.yml, 如下:

```
[root@kolla ~]# vi /etc/kolla/globals.yml
openstack_release: "4.0.0"           //4.0.0对应Ocata版本
kolla_internal_vip_address: "192.168.125.100" //虚拟IP地址
network_interface: "eno50332184"      //Openstack服务监听的地址接口
neutron_external_interface: "eno16777736" //Neutron网络使用的桥接网口
```

上述配置中, 为 kolla_internal_vip_address 指定的 IP 地址为虚拟 IP 地址, 其与 eno50332184 网口的 IP 地址处于同一个网段, 在 All-In-One 部署中没有任何意义, eno16777736 为 Neutron 桥接所使用的物理网口, Nova 虚拟机将通过此接口与外部通信。部署配置文件修改完成后, 使用 kolla-ansible 命令检查配置是否正确, 如下:

```
kolla-ansible prechecks -i /root/all-in-one
```

如果配置文件存在语法上的问题, 或者 OpenStack 服务所需的资源得不到满足, 则 prechecks 命令将失败, 在部署之前必须保证检查结果完全成功。检查通过后, 使用 kolla-ansible 命令进行一键部署, 如下:

```
kolla-ansible deploy -i /root/all-in-one
```

部署成功之后，通过 Docker 命令查看已启动运行的容器，如下：

```
[root@kolla ~]# docker ps
```

CONTAINER ID	COMMAND	CREATED	STATUS	NAMES
1f3f2ecaef9f	"kolla_start"	2 days ago	Up 21 hours	horizon
91c63fae7f49	"kolla_start"	2 days ago	Up 21 hours	heat_engine
98159c36b089	"kolla_start"	2 days ago	Up 21 hours	heat_api_cfn
aae2a0f21386	"kolla_start"	2 days ago	Up 21 hours	heat_api
cc77bc374f59	"kolla_start"	2 days ago	Up 21 hours	neutron_metadata_agent
18b8f794fef9	"kolla_start"	2 days ago	Up 21 hours	neutron_l3_agent

部署完成且 OpenStack 服务容器启动之后，所有启动的 OpenStack 及其基础组件服务的配置文件均在 /etc/kolla 目录下，如下所示：

```
[root@kolla ~]# cd /etc/kolla
[root@kolla kolla]# ls -l
total 88
drwxr-xr-x. 3 root root    40 Feb 21 23:32 cron
drwxr-xr-x. 6 root root    95 Feb 21 23:33 fluentd
drwxr-xr-x. 2 root root    46 Feb 21 23:35 glance-api
drwxr-xr-x. 2 root root    51 Feb 21 23:35 glance-registry
-rw-r--r--. 1 root root 11216 Feb 21 23:05 globals.yml
drwxr-xr-x. 2 root root    42 Feb 21 23:33 haproxy
drwxr-xr-x. 2 root root    40 Feb 21 23:41 heat-api
.....
```

如果用户需要更改对应服务的配置，则只需到 /etc/kolla 目录中找到对应的项目子目录并修改配置文件即可，例如因为本节中 OpenStack 部署在虚拟机上，如果虚拟机不支持 KVM 虚拟化，则需要修改虚拟类型为 qemu，如下：

```
vim /etc/kolla/nova-compute/nova.conf
[libvirt]
...
virt_type=qemu
```

至此，基于 Docker 容器的 OpenStack 部署已经完成，全部 OpenStack 及其基础组件均以 Docker 容器形式运行在主机上，下一节将对 Kolla 部署的 OpenStack 进行功能测试。

15.3.4 OpenStack 功能验证

Kolla-ansible 除了能够通过 Docker 容器形式部署 OpenStack，还提供了一些命令行工具来帮助用户使用和管理 OpenStack 集群，例如用户可以通过 post-deploy 命令来生成 openrc 文件，生成的 openrc 文件默认存储在 /etc/kolla 目录中，如下：

```
[root@kolla ~]# kolla-ansible post-deploy
[root@kolla ~]#more /etc/kolla/admin-openrc.sh
```



```

export OS_PROJECT_DOMAIN_NAME=default
export OS_USER_DOMAIN_NAME=default
export OS_PROJECT_NAME=admin
export OS_TENANT_NAME=admin
export OS_USERNAME=admin
export OS_PASSWORD=admin
export OS_AUTH_URL=http://192.168.125.100:35357/v3
export OS_IDENTITY_API_VERSION=3

```

此外，Kolla-ansible 在源代码的 kolla-ansible/tools 目录中还提供了一个名为 init-runonce 的 sh 脚本来帮助用户初始化 OpenStack 集群，其功能包括下载 cirros 镜像并上传至 Glance，以及创建几个租户网络并设置 Nova 的 quota，用户需要根据自己的网络环境对 kolla-ansible/tools/init-runonce 进行修改，通常建议将此脚本拷贝到当前部署目录再修改，如下：

```

[root@kolla ~]#cpkolla-ansible/tools/init-runonce .
[root@kolla ~]#viinit-runonce
.....
#IMAGE_URL=http://download.cirros-cloud.net/0.3.4/
IMAGE_URL=http://images.trystack.cn/0.3.4/
IMAGE=cirros-0.3.4-x86_64-disk.img
IMAGE_NAME=cirros
EXT_NET_CIDR='10.20.30.0/24'
EXT_NET_RANGE='start=10.20.30.100,end=10.20.30.200'
EXT_NET_GATEWAY='10.20.30.2'
.....

```

上述修改中，将 Cirros 镜像的下载地址改为国内九州云的镜像下载地址，这样下载过程会非常快。此外，外部网络“EXT_NET_CIDR=10.20.30.0/24”为实验环境中可以访问外网的网络，此处需要根据用户各自的网络环境进行设置，而“EXT_NET_GATEWAY=10.20.30.2”为实验环境中的外网网关，该参数也需要根据实际网络情况进行设置。设置完成之后，即可运行该脚本初始化 OpenStack，如下：

```

[root@kolla ~]#source /etc/kolla/admin-openrc.sh
[root@kolla ~]#chmod 755 init-runonce
[root@kolla ~]#bashinit-runonce

```

运行完成后，对 OpenStack 进行简单的验证，如下：

//查看默认创建的用户

```
[root@kolla ~]# openstack user list
```

ID	Name
00b7c43f9d424b58ae9e4a88cd63c1f5	neutron
01e4c05de74b4c43b42f7b178a45c339	heat
10a54ddb21b84c298c74b38b6fbf3db4	placement
6c90eec0e03b420b8e0784de8df6a89c	nova
80b69d73565b4626b1b901a99be3ec44	glance
8d92e37b0ffd43f4babf045ddcccf54b	admin

```
| a713d2766b9d4fe39f9b2a4e22d77922 | heat_domain_admin |
+-----+-----+
//查看默认创建的OpenStack服务
[root@kolla ~]# openstack service list
+-----+-----+-----+
| ID | Name | Type |
+-----+-----+-----+
| 07b3c3c9bb234513a5b8f07582c637bf | glance | image |
| 1b87f715af834a2fb5b3e01396b16b29 | heat | orchestration |
| 27a396cf7c504d97a219b0cf78b1e1ba | nova_legacy | compute_legacy |
| 87261695a6ea48e1b292f28fb2c51da6 | placement | placement |
| a7d0659432164bb481fc04a3c583aee0 | nova | compute |
| acea2508cad64c0fac79fe374037b65a | heat-cfn | cloudformation |
| b36f23d929fa44899f21004d24184fb0 | keystone | identity |
| fc64016e2fab4140a995b2fa41da76de | neutron | network |
+-----+-----+-----+
//查看初始化过程中创建的镜像
[root@kolla ~]# openstack image list
+-----+-----+-----+
| ID | Name | Status |
+-----+-----+-----+
| 3b337b3b-0acc-4f88-9f17-fa805b19a2e0 | cirros | active |
+-----+-----+-----+
//查看初始化过程中创建的网络
[root@kolla ~]# openstack network list
+-----+-----+-----+
| ID | Name | Subnets |
+-----+-----+-----+
| ...b7871 | demo-net | e04a1a4a-9ba3-4e82-aedd-4d48fce809d3 |
| ...3046b | public1 | 5a776f1f-e2c4-4038-bcd5-557847caf3eb |
+-----+-----+-----+
[root@kolla ~]# openstack subnet list
+-----+-----+-----+
| ID | Name | Network | Subnet |
+-----+-----+-----+
| ...3eb | public1-subnet | cd4c3498-6132-4171-9a97-5ede5283046b | 10.20.30.0/24 |
| ...9d3 | demo-subnet | 680f6993-cc02-4a9a-a16b-3dda928b7871 | 10.0.0.0/24 |
+-----+-----+-----+
[root@kolla ~]# openstack router list
+-----+-----+-----+
| ID | Name | Status | State | Distributed | HA | Project |
+-----+-----+-----+
| ...87b6 | demo-router | ACTIVE | UP | False | False | ...4e17c78 |
+-----+-----+-----+
```

此外，kolla-ansible/tools/init-runonce 脚本执行完成后，会有一个创建虚拟机的提示命令，运行该命令即可在 Openstack 中创建一个虚拟机，如下：

```
[root@kolla ~]# openstack server create \
--image cirros \
--flavor ml.tiny \
```

```
--key-name mykey \
--nic net-id=680f6993-cc02-4a9a-a16b-3dda928b7871 \
demo1
```

上述命令执行完成后，检查虚拟机是否启动成功，如下：

```
[root@kolla ~]# openstack server list
```

ID	Name	Status	Networks	Image Name
...b02cd594d	demo1	ACTIVE	demo-net=10.0.0.6	cirros

此时，可以通过 <http://192.168.125.10> 来登录 OpenStack 的 Dashboard，注意登录 IP 地址不是虚拟 IP，而是 eno50332184 接口上的 IP，即服务监听 IP 地址，如图 15-13 所示。

登录 Dashboard 后，用户可以为刚创建的虚拟机分配 FloatingIP 地址，或者进行其他功能的测试，分配 FloatingIP 之后的实例如图 15-14 所示。

通过上述测试可以看到，通过 Kolla 和 Kolla-ansible 项目部署 OpenStack 非常简单，总结起来，就是使用 Kolla 编译 Docker 镜像，使用 Kolla-ansible 部署编译好的 Docker 镜像，当所有封装了 OpenStack 服务的 Docker 容器正常启动后，用户便可正常使用 OpenStack 提供的云计算资源。

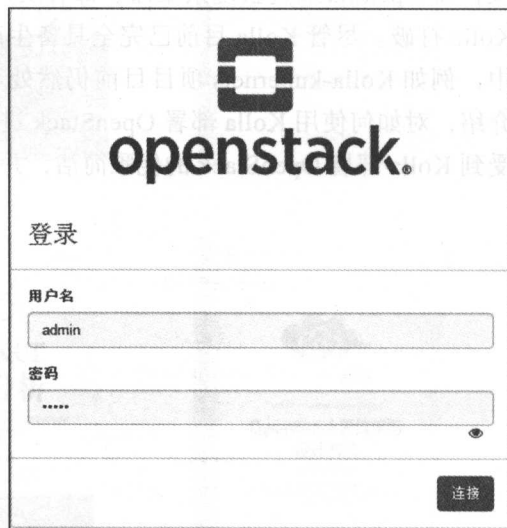


图 15-13 Ocata 版本 Dashboard 登录界面

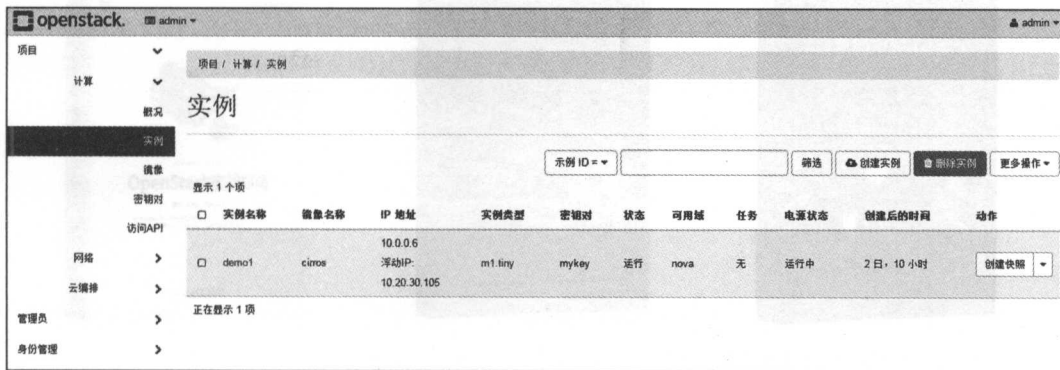
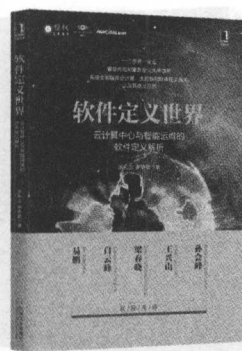
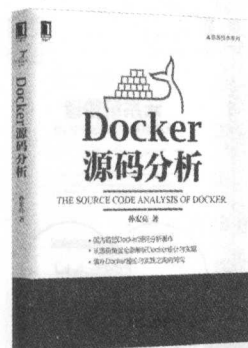
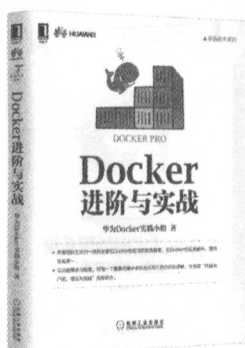
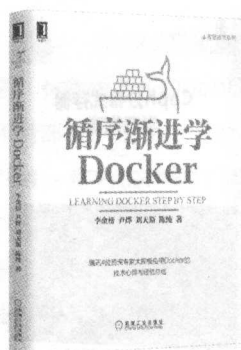
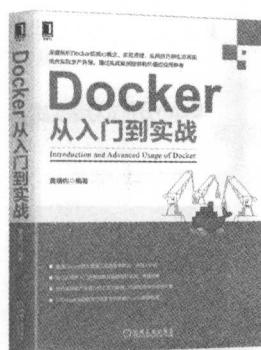


图 15-14 具有浮动 IP 的实例

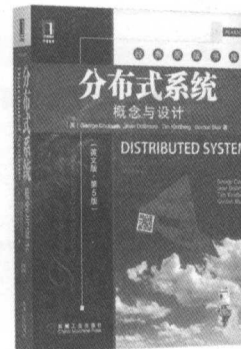
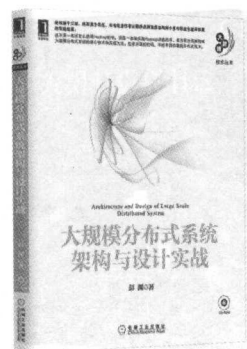
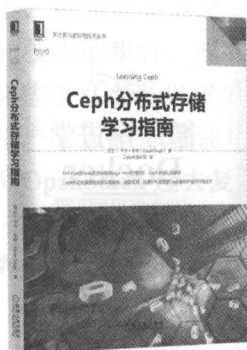
15.4 本章小结

自从 OpenStack 的 Liberty 版本以 Big Tent 模式支持和应对 Docker 容器的发展以来, OpenStack 社区涌现出了诸多与 Docker 相关的孵化项目, 而在 OpenStack 刚刚发行的 Ocata 版本中, 最大的亮点之一便是更加友好地支持和拥抱 Docker 容器技术。可以预测, 与 Docker 技术的集合, 或者说以二者互补为出发点的 OpenStack 项目必将引领社区的发展, 同时这些项目也必将成为 OpenStack 社区的活跃项目。本章介绍的 Kolla 项目尽管目前还未被用户大面积部署使用, 但是从社区活跃程度来看, Kolla 在 2017 年必将得到极大普及, 而 OpenStack 过去复杂难懂、部署及升级维护困难的局面也将被携 Docker 之长而来的 Kolla 打破。尽管 Kolla 目前已完全具备生产环境部署的能力, 但是也还在不断地优化完善中, 例如 Kolla-kubernetes 项目目前仍然处于测试评估阶段。本章对 Kolla 项目进行了详细介绍, 对如何使用 Kolla 部署 OpenStack 进行了讲解, 通过对本章的学习, 读者应该可以感受到 Kolla 部署 OpenStack 的优雅简洁, 并能够通过 Kolla 部署自己的 OpenStack 集群。

推荐阅读



推荐阅读



作者简介

山金孝（Warrior）国内较早接触OpenStack的一线技术专家，长期致力于OpenStack的研究、实践和生产环境部署，是OpenStack社区的积极参与者和实践者。作为由传统IT架构转型为云计算领域的技术专家，参与并设计实施了移动、电信、联通、招行、国家电网和长安汽车等多家大中型国有企业的高可用业务系统，在系统容灾和高可用集群建设上具有多年的项目实施经验。

曾就职于IBM，现就职于招商银行，主持设计并实施了招商银行重庆分行的OpenStack高可用生产系统集群，目前是招商银行重庆分行核心业务系统和云计算基础架构平台的主要负责人。

此外，他还是IBM认证的高级技术专家和DB2方向的高级DBA，同时也是RedHat认证的Linux系统工程师。

OpenStack Cluster HA

Deployment and Operation

本书是对OpenStack高可用集群部署和实现的多维深度实践，总结了OpenStack高可用的不同方案，并详细讲解了计算、存储和网络各个模块的高可用架构及实施。难能可贵的是，本书没有停留在理论和实验环境层面，而是总结了大量生产环境的实践。

—— 肖力 云技术社区创始人

金孝具有多年金融行业及大型制造业的云计算从业经验，经历过诸多大中型企业的核心系统项目建设，在云计算及虚拟化方面积累了多年的项目经验，也是国内较早一批接触OpenStack并对其进行研究和部署实践的开拓者。这是一本真正由OpenStack终端用户编写，并且面向生产环境部署的专著，书中有大量代码和实施步骤，相信对OpenStack的落地和运维能够起到积极的推动作用。

—— 张鹏 IBM全球技术服务部高级工程师/客户服务经理

长久以来，一直期望有一本全方位讲解OpenStack高可用部署与实施的图书，让更多的工程师能够理解、掌握和实施面向生产系统的OpenStack云计算项目。很欣慰能够看到本书的面世，它从理论到实战部署，再到运维，全方位讲解了OpenStack的高可用集群。

—— 刁坤华 重庆奇梦达科技有限公司创始人

作者具有多年OpenStack的项目实施经验，本书采用理论与实践相结合的方式，由浅入深地讲解了在生产系统中部署OpenStack高可用集群的方法，总结了实践中常见问题的解决方案，是为即将和正在使用OpenStack的云计算工程师准备的“核武器”。

—— 宋珩 招商银行重庆分行信息技术部总经理

金孝具备深厚的理论功底和丰富的行业实战经验，不同于一般作者，他开展了非常多的系统性的工程实践，积累了丰富的实战经验。他一直致力于研究最新的云计算技术，始终奋斗在最前沿，做出了很多卓有成效的探索和实践。本书理论与实践相结合，非常适合OpenStack初学者、架构师、运维工程师等人员阅读。相信您从本书中一定能有宝贵的收获。

—— 周鹏 招商银行信息技术部高级工程师



投稿热线: (010) 88379604
客服热线: (010) 88379426 88361066
购书热线: (010) 68326294 88379649 68995259

华章网站: www.hzbook.com
网上购书: www.china-pub.com
数字阅读: www.hzmedia.com.cn

上架指导: 计算机/云计算

ISBN 978-7-111-58095-9



9 787111 580959 >

定价: 79.00元